# 第三版：并发编程 80 道

## 为什么要使用并发编程

1、 提升多核 CPU 的利用率：一般来说一台主机上的会有多个 CPU 核心，我们可以创建多个线程，理论上讲操作系统可以将多个线程分配给不同的 CPU 去执行，每个 CPU 执行一个线程，这样就提高了 CPU 的使用效率，如果使用单线程就只能有一个 CPU 核心被使用。

2、 比如当我们在网上购物时，为了提升响应速度，需要拆分，减库存，生成订单等等这些操作，就可以进行拆分利用多线程的技术完成。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 。

**简单来说就是：**

1、 充分利用多核 CPU 的计算能力；

2、 方便进行业务拆分，提升应用性能

## 多线程应用场景

**例如：**

迅雷多线程下载、数据库连接池、分批发送短信等。

## 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。

## 并发编程三个必要因素是什么?

**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。

**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）

**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）

## 在 Java 程序中怎么保证多线程的运行安全?

**出现线程安全问题的原因一般都是三个原因：**

**1、** 线程切换带来的原子性问题 解决办法：使用多线程之间同步 synchronized 或使用锁(lock)。

**2、** 缓存导致的可见性问题 解决办法：synchronized、volatile、LOCK，可以解决可见性问题

3、 编译优化带来的有序性问题 解决办法：Happens-Before 规则可以解决有序性问题

## 并行和并发有什么区别?

1、 并发：多个任务在同一个 CPU 核上，按细分的时间片轮流(交替)执行，从逻辑上来看那些任务是同时执行。

2、 并行：单位时间内，多个处理器或多核处理器同时处理多个任务，是真正意义上的“同时进行”。

3、 串行：有 n 个任务，由一个线程按顺序执行。由于任务、方法都在一个线程执行所以不存在线程不安全情况，也就不存在临界区的问题。

**做一个形象的比喻：**

1、 并发 = 俩个人用一台电脑。

2、 并行 = 俩个人分配了俩台电脑。

3、 串行 = 俩个人排队使用一台电脑。

## 什么是多线程

**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。**

## 线程和进程区别

什么是线程和进程？

**进程**

一个在内存中运行的应用程序。 每个正在系统上运行的程序都是一个进程

**线程**

进程中的一个执行任务（控制单元）， 它负责在程序里独立执行。

一个进程至少有一个线程，一个进程可以运行多个线程，多个线程可共享数据

**进程与线程的区别**

1、 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

2、 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。

3、 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。

4、 内存分配：同一进程的线程共享本进程的地址空间和资源，而进程与进程之间的地址空间和资源是相互独立的

5、 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，但是一个线程崩溃有可能导致整个进程都死掉。所以多进程要比多线程健壮。

6、 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?

1、 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。

**2、** 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。

**3、** 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。

**4、** 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。

## 守护线程和用户线程有什么区别呢?

**用户（User）线程：**

运行在前台，执行具体的任务，如程序的主线程、连接网络的子线程等都是用户线程

**守护（Daemon）线程：**

运行在后台，为其他前台线程服务。也可以说守护线程是 JVM 中非守护线程的“佣人”。一旦所有用户线程都结束运行，守护线程会随 JVM 一起结束工作

## 如何在 Windows 和 Linux 上查找哪个线程 cpu 利用率最高?

windows 上面用任务管理器看，linux 下可以用 top 这个工具看。

**1、** 找出 cpu 耗用厉害的进程 pid， 终端执行 top 命令，然后按下 shift+p (shift+m 是找出消耗内存最高)查找出 cpu 利用最厉害的 pid 号

**2、** 根据上面第一步拿到的 pid 号，top -H -p pid 。然后按下 shift+p，查找出 cpu 利用率最厉害的线程号，比如 top -H -p 1328

**3、** 将获取到的线程号转换成 16 进制，去百度转换一下就行

**4、** 使用 jstack 工具将进程信息打印输出，jstack pid 号 > /tmp/t.dat，比如 jstack 31365 > /tmp/t.dat

**5、** 编辑/tmp/t.dat 文件，查找线程号对应的信息

或者直接使用 JDK 自带的工具查看“jconsole”、“visualVm”，这都是 JDK 自带的，可以直接在 JDK 的 bin 目录下找到直接使用

## 什么是线程死锁

**1、** 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）。

**2、** 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。

**3、** 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。

## 形成死锁的四个必要条件是什么

1、 互斥条件：在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，就只能等待，直至占有资源的进程用毕释放。

2、 占有且等待条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。

3、 不可抢占条件：别人已经占有了某项资源，你不能因为自己也需要该资源，就去把别人的资源抢过来。

4、 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。（比如一个进程集合，A 在等 B，B 在等 C，C 在等 A）

## 如何避免线程死锁

1、 避免一个线程同时获得多个锁

2、 避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源

3、 尝试使用定时锁，使用 lock.tryLock(timeout)来替代使用内部锁机制

## 创建线程的四种方式

**继承 Thread 类；**

```
public class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " run()
方法正在执行...");
}
```

**实现 Runnable 接口；**

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName() + " run()
方法执行中...");
}
```

**实现 Callable 接口；**

```
public class MyCallable implements Callable<Integer> {

@Override
public Integer call() {
    System.out.println(Thread.currentThread().getName() + " call()方法
执行中...");
    return 1;
}
```

**使用匿名内部类方式**

```
public class CreateRunnable {
```

```
    public static void main(String[] args) {
        //创建多线程创建开始
        Thread thread = new Thread(new Runnable() {
                    public void run() {
                    for (int i = 0; i < 10; i++) {
                    System.out.println("i:" + i);
                }
            }
        });
        thread.start();
    }
}
```

## 说一下 runnable 和 callable 有什么区别

**相同点：**

**1、** 都是接口

**2、** 都可以编写多线程程序

**3、** 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。

## 线程的 run()和 start()有什么区别?

**1、** 每个线程都是通过某个特定 Thread 对象所对应的方法 run()来完成其操作的，run()方法称为线程体。通过调用 Thread 类的 start()方法来启动一个线程。

**2、** start() 方法用于启动线程，run() 方法用于执行线程的运行时代码。run() 可以重复调用，而 start() 只能调用一次。

**3、** start()方法来启动一个线程，真正实现了多线程运行。调用 start()方法无需等待 run 方法体代码执行完毕，可以直接继续执行其他的代码； 此时线程是处于就绪状态，并没有运行。然后通过此 Thread 类调用方法 run()来完成其运行状态，run()方法运行结束， 此线程终止。然后 CPU 再调度其它线程。

**4、** run()方法是在本线程里的，只是线程里的一个函数，而不是多线程的。 如果直接调用 run()，其实就相当于是调用了一个普通函数而已，直接待用 run()方法必须等待 run()方法执行完毕才能执行下面的代码，所以执行路径还是只有一条，根本就没有线程的特征，所以在多线程执行时要使用 start()方法而不是 run()方法。

## 为什么我们调用 start() 方法时会执行 run() 方法，为什么我们不能直接调用 run() 方法?

这是另一个非常经典的 java 多线程面试问题，而且在面试中会经常被问到。很简单，但是很多人都会答不上来!

new 一个 Thread，线程进入了新建状态。调用 start() 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。 start() 会执行线程的相应准备工作，然后自动执行 run() 方法的内容，这是真正的多线程工作。

而直接执行 run() 方法，会把 run 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。

**总结：**

调用 start 方法方可启动线程并使线程进入就绪状态，而 run 方法只是 thread 的一个普通方法调用，还是在主线程里执行。

## 什么是 Callable 和 Future?

Callable 接口类似于 Runnable，从名字就可以看出来了，但是 Runnable 不会返回结果，并且无法抛出返回结果的异常，而 Callable 功能更强大一些，被线程执行后，可以返回值，这个返回值可以被 Future 拿到，也就是说，Future 可以拿到异步执行任务的返回值。

Future 接口表示异步任务，是一个可能还没有完成的异步任务的结果。所以说 Callable 用于产生结果，Future 用于获取结果。

## 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。

## 线程的状态

**1、** 新建(new)：新创建了一个线程对象。

**2、** 就绪（可运行状态）(runnable)：线程对象创建后，当调用线程对象的 start()方法，该线程处于就绪状态，等待被线程调度选中，获取 cpu 的使用权。

**3、**运行(running):可运行状态(runnable)的线程获得了 cpu 时间片（timeslice），执行程序代码。注：就绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；

**4、** 阻塞(block)：处于运行状态中的线程由于某种原因，暂时放弃对 CPU 的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才 有机会再次被 CPU 调用以进入到运行状态。

**阻塞的情况分三种：**

**1、** 等待阻塞：

运行状态中的线程执行 wait()方法，JVM 会把该线程放入等待队列(waitting queue)中，使本线程进入到等待阻塞状态；

**2、** 同步阻塞：

线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)，，则 JVM 会把该线程放入锁池(lock pool)中，线程会进入同步阻塞状态；

**3、** 其他阻塞:

通过调用线程的 sleep()或 join()或发出了 I/O 请求时,线程会进入到阻塞状态。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，线程重新转入就绪状态。

**4、** 死亡(dead)(结束):

线程 run()、main()方法执行结束，或者因异常退出了 run()方法，则该线程结束生命周期。死亡的线程不可再次复生。

## Java 中用到的线程调度算法是什么?

计算机通常只有一个 CPU，在任意时刻只能执行一条机器指令，每个线程只有获得 CPU 的使用权才能执行指令。所谓多线程的并发运行，其实是指从宏观上看，各个线程轮流获得 CPU 的使用权，分别执行各自的任务。在运行池中，会有多个处于就绪状态的线程在等待 CPU，JAVA 虚拟机的一项任务就是负责线程的调度,线程调度是指按照特定机制为多个线程分配 CPU 的使用权。(Java 是由 JVM 中的线程计数器来实现线程调度)

**有两种调度模型:**

分时调度模型和抢占式调度模型。

**1、** 分时调度模型是指让所有的线程轮流获得 cpu 的使用权，并且平均分配每个线程占用的 CPU 的时间片这个也比较好理解。

**2、** Java 虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用 CPU，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用 CPU。处于运行状态的线程会一直运行，直至它不得不放弃 CPU。

## 线程的调度策略

线程调度器选择优先级最高的线程运行，但是，如果发生以下情况，就会终止线程的运行

1、 线程体中调用了 yield 方法让出了对 cpu 的占用权利

2、 线程体中调用了 sleep 方法使线程进入睡眠状态

3、 线程由于 IO 操作受到阻塞

4、 另外一个更高优先级线程出现

5、 在支持时间片的系统中，该线程的时间片用完

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

1、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。

2、 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。

3、 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）。

## 请说出与线程同步以及线程调度相关的方法。

1、 wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；

2、 sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理 InterruptedException 异常；

3、 notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由 JVM 确定唤醒哪个线程，而且与优先级无关；

4、 notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

## sleep() 和 wait() 有什么区别?

两者都可以暂停线程的执行

1、 类的不同：

sleep() 是 Thread 线程类的静态方法，wait() 是 Object 类的方法。

2、 是否释放锁：

sleep() 不释放锁；wait() 释放锁。

3、 用途不同：

Wait 通常被用于线程间交互/通信，sleep 通常被用于暂停执行。

4、 用法不同：

wait() 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 notify() 或者 notifyAll() 方法。sleep() 方法执行完成后，线程会自动苏醒。或者可以使用 wait(long timeout)超时后线程会自动苏醒。

## 你是如何调用 wait() 方法的？使用 if 块还是循环？为什么？

处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件的情况下退出。

wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。下面是一段标准的使用 wait 和 notify 方法的代码：

```
synchronized (monitor) {
    //  判断条件谓词是否得到满足
    while(!locked) {
    //  等待唤醒
    monitor.wait();
    }
    //  处理其他的业务逻辑
}
```

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

因为 Java 所有类的都继承了 Object，Java 想让任何对象都可以作为锁，并且 wait()，notify()等方法用于等待对象的锁或者唤醒线程，在 Java 的线程中并没有可供任何对象使用的锁，所以任意对象调用方法一定定义在 Object 类中。

有的人会说，既然是线程放弃对象锁，那也可以把 wait()定义在 Thread 类里面啊，新定义的线程继承于 Thread 类，也不需要重新定义 wait()方法的实现。然而，这样做有一个非常大的问题，一个线程完全可以持有很多锁，你一个线程放弃锁的时候，到底要放弃哪个锁？当然了，这种设计并不是不能实现，只是管理起来更加复杂。

## 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。

## Thread 类中的 yield 方法有什么作用?

使当前线程从执行状态（运行状态）变为可执行态（就绪状态）。

当前线程到了就绪状态，那么接下来哪个线程会从就绪状态变成执行状态呢？可能是当前线程，也可能是其他线程，看系统的分配了。

## 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。

## 线程的 sleep()方法和 yield()方法有什么区别?

**1、** sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；

**2、** 线程执行 sleep()方法后转入阻塞（blocked）状态，而执行 yield()方法后转入就绪（ready）状态；

**3、** sleep()方法声明抛出 InterruptedException，而 yield()方法没有声明任何异常；

**4、** sleep()方法比 yield()方法（跟操作系统 CPU 调度相关）具有更好的可移植性，通常不建议使用 yield()方法来控制并发线程的执行。

## 如何停止一个正在运行的线程?

在 java 中有以下 3 种方法可以终止正在运行的线程：

**1、** 使用退出标志，使线程正常退出，也就是当 run 方法完成后线程终止。

2、 使用 stop 方法强行终止，但是不推荐这个方法，因为 stop 和 suspend 及 resume 一样都是过期作废的方法。

3、 使用 interrupt 方法中断线程。

## Java 中 interrupted 和 isInterrupted 方法的区别?

**interrupt:**

用于中断线程。调用该方法的线程的状态为将被置为”中断”状态。

注意：线程中断仅仅是置线程的中断状态位，不会停止线程。需要用户自己去监视线程的状态为并做处理。支持线程中断的方法（也就是线程中断后会抛出 interruptedException 的方法）就是在监视线程的中断状态，一旦线程的中断状态被置为“中断状态”，就会抛出中断异常。

**interrupted:**

是静态方法，查看当前中断信号是 true 还是 false 并且清除中断信号。如果一个线程被中断了，第一次调用 interrupted 则返回 true，第二次和后面的就返回 false 了。

**isInterrupted:**

是可以返回当前中断信号是 true 还是 false，与 interrupt 最大的差别

## 什么是阻塞式方法?

阻塞式方法是指程序会一直等待该方法完成期间不做其他事情，ServerSocket 的 accept()方法就是一直等待客户端连接。这里的阻塞是指调用结果返回之前，当前线程会被挂起，直到得到结果之后才会返回。此外，还有异步和非阻塞式方法在任务完成前就返回。

## Java 中你怎样唤醒一个阻塞的线程?

首先 ，wait()、notify() 方法是针对对象的，调用任意对象的 wait()方法都将导致线程阻塞，阻塞的同时也将释放该对象的锁，相应地，调用任意对象的 notify()方法则将随机解除该对象阻塞的线程，但它需要重新获取该对象的锁，直到获取成功才能往下执行；

其次，wait、notify 方法必须在 synchronized 块或方法中被调用，并且要保证同步块或方法的锁对象与调用 wait、notify 方法的对象是同一个，如此一来在调用 wait 之前当前线程就已经成功获取某对象的锁，执行 wait 阻塞后当前线程就将之前获取的对象锁释放。

## notify() 和 notifyAll() 有什么区别?

**1、** 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。

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

**3、** 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。

## 如何在两个线程间共享数据?

**在两个线程间共享变量即可实现共享**

一般来说，共享变量要求变量本身是线程安全的，然后在线程内使用的时候，如果有对共享变量的复合操作，那么也得保证复合操作的线程安全性

## Java 如何实现多线程之间的通讯和协作?

可以通过中断 和 共享变量的方式实现线程间的通讯和协作

比如说最经典的生产者-消费者模型：当队列满时，生产者需要等待队列有空间才能继续往里面放入商品，而在等待的期间内，生产者必须释放对临界资源（即队列）的占用权。因为生产者如果不释放对临界资源的占用权，那么消费者就无法消费队列中的商品，就不会让队列有空间，那么生产者就会一直无限等待下去。因此，一般情况下，当队列满时，会让生产者交出对临界资源的占用权，并进入挂起状态。然后等待消费者消费了商品，然后消费者通知生产者队列有空间了。同样地，当队列空时，消费者也必须等待，等待生产者通知它队列中有商品了。这种互相通信的过程就是线程间的协作。

**Java 中线程通信协作的最常见方式：**

1、 syncrhoized 加锁的线程的 Object 类的 wait()/notify()/notifyAll()

2、 ReentrantLock 类加锁的线程的 Condition 类的 await()/signal()/signalAll()

**线程间直接的数据交换：**

通过管道进行线程间通信：字节流、字符流

## 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。

同步块更要符合开放调用的原则，只在需要锁住的代码块锁住相应的对象，这样从侧面来说也可以避免死锁。

请知道一条原则：同步的范围越小越好。

## 什么是线程同步和线程互斥，有哪几种实现方式?

**1、** 当一个线程对共享的数据进行操作时，应使之成为一个”原子操作“，即在没有完成相关操作之前，不允许其他线程打断它，否则，就会破坏数据的完整性，必然会得到错误的处理结果，这就是线程的同步。

**2、** 在多线程应用中，考虑不同线程之间的数据同步和防止死锁。当两个或多个线程之间同时等待对方释放资源的时候就会形成线程之间的死锁。为了防止死锁的发生，需要通过同步来实现线程安全。

**3、** 线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。

**4、** 线程间的同步方法大体可分为两类：用户模式和内核模式。顾名思义，内核模式就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态，而用户模式就是不需要切换到内核态，只在用户态完成操作。

**5、** 用户模式下的方法有：

原子操作（例如一个单一的全局变量），临界区。内核模式下的方法有：事件，信号量，互斥量。

**实现线程同步的方法**

**1、** 同步代码方法：sychronized 关键字修饰的方法

**2、** 同步代码块：sychronized 关键字修饰的代码块

**3、** 使用特殊变量域 volatile 实现线程同步：volatile 关键字为域变量的访问提供了一种免锁机制

**4、** 使用重入锁实现线程同步：reentrantlock 类是可冲入、互斥、实现了 lock 接口的锁他与 sychronized 方法具有相同的基本行为和语义

## 在监视器(Monitor)内部，是如何做线程同步的？程序应该做哪种级别的同步？

在 java 虚拟机中，监视器和锁在 Java 虚拟机中是一块使用的。监视器监视一块同步代码块，确保一次只有一个线程执行同步代码块。每一个监视器都和一个对象引用相关联。线程在获取锁之前不允许执行同步代码。

一旦方法或者代码块被 synchronized 修饰，那么这个部分就放入了监视器的监视区域，确保一次只能有一个线程执行该部分的代码，线程在获取锁之前不允许执行该部分的代码

另外 java 还提供了显式监视器( Lock )和隐式监视器( synchronized )两种锁方案

## 如果你提交任务时，线程池队列已满，这时会发生什么

**有俩种可能：**

**1、** 如果使用的是无界队列 LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存放任务

**2、** 如果使用的是有界队列比如 ArrayBlockingQueue，任务首先会被添加到 ArrayBlockingQueue 中，ArrayBlockingQueue 满了，会根据 maximumPoolSize 的值增加线程数量，如果增加了线程数量还是处理不过来，ArrayBlockingQueue 继续满，那么则会使用拒绝策略 RejectedExecutionHandler 处理满了的任务，默认是 AbortPolicy

## 什么叫线程安全？servlet 是线程安全吗？

**1、** 线程安全是编程中的术语，指某个方法在多线程环境中被调用时，能够正确地处理多个线程之间的共享变量，使程序功能正确完成。

**2、** Servlet 不是线程安全的，servlet 是单实例多线程的，当多个线程同时访问同一个方法，是不能保证共享变量的线程安全性的。

**3、** Struts2 的 action 是多实例多线程的，是线程安全的，每个请求过来都会 new 一个新的 action 分配给这个请求，请求完成后销毁。

**4、** SpringMVC 的 Controller 是线程安全的吗？不是的，和 Servlet 类似的处理流程。

**5、** Struts2 好处是不用考虑线程安全问题；Servlet 和 SpringMVC 需要考虑线程安全问题，但是性能可以提升不用处理太多的 gc，可以使用 ThreadLocal 来处理多线程的问题。

## 在 Java 程序中怎么保证多线程的运行安全?

**1、** 使用安全类，比如 java.util.concurrent 下的类，使用原子类 AtomicInteger

**2、** 使用自动锁 synchronized。

**3、** 使用手动锁 Lock。

**手动锁 Java 示例代码如下：**

```
Lock lock = new ReentrantLock();
    lock、lock();

    try {
        System、out、println("获得锁");
            } catch (Exception e) {
            // TODO: handle exception
    } finally {
    System、out、println("释放锁");
```

```
    lock、unlock();
}
```

## 你对线程优先级的理解是什么?

**1、** 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。

**2、** Java 的线程优先级调度会委托给操作系统去处理，所以与具体的操作系统优先级有关，如非特别需要，一般无需设置线程优先级。

**3、** 当然，如果你真的想设置优先级可以通过 setPriority()方法设置，但是设置了不一定会该变，这个是不准确的

## 线程类的构造方法、静态块是被哪个线程调用的

**1、** 这是一个非常刁钻和狡猾的问题。请记住：线程类的构造方法、静态块是被 new 这个线程类所在的线程所调用的，而 run 方法里面的代码才是被线程自身所调用的。

**2、** 如果说上面的说法让你感到困惑，那么我举个例子，假设 Thread2 中 new 了 Thread1，main 函数中 new 了 Thread2，那么：

Thread2 的构造方法、静态块是 main 线程调用的，Thread2 的 run()方法是 Thread2 自己调用的

Thread1 的构造方法、静态块是 Thread2 调用的，Thread1 的 run()方法是 Thread1 自己调用的

## Java 中怎么获取一份线程 dump 文件？你如何在 Java 中获取线程堆栈？

**1、** Dump 文件是进程的内存镜像。可以把程序的执行状态通过调试器保存到 dump 文件中。

**2、** 在 Linux 下，你可以通过命令 kill -3 PID （Java 进程的进程 ID）来获取 Java 应用的 dump 文件。

**3、** 在 Windows 下，你可以按下 Ctrl + Break 来获取。这样 JVM 就会将线程的 dump 文件打印到标准输出或错误文件中，它可能打印在控制台或者日志文件中，具体位置依赖应用的配置。

## 一个线程运行时发生异常会怎样？

如果异常没有被捕获该线程将会停止执行。Thread.UncaughtExceptionHandler 是用于处理未捕获异常造成线程突然中断情况的一个内嵌接口。当一个未捕获异常将造成线程中断的时候，JVM 会使用 Thread.getUncaughtExceptionHandler()来查询线程的 UncaughtExceptionHandler 并将线程和异常作为参数传递给 handler 的 uncaughtException()方法进行处理。

## Java 线程数过多会造成什么异常？

1、 线程的生命周期开销非常高

2、 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。

## 多线程的常用方法

| 方法 名 | 描述 |
| --- | --- |
| sleep() | 强迫一个线程睡眠N毫秒 |
| isAlive() | 判断一个线程是否存活。 |
| join() | 等待线程终止。 |
| activeCount() | 程序中活跃的线程数。 |
| enumerate() | 枚举程序中的线程。 |
| currentThread() | 得到当前线程。 |
| isDaemon() | 一个线程是否为守护线程。 |
| setDaemon() | 设置一个线程为守护线程。 |
| setName() | 为线程设置一个名称。 |

| | |
|---|---|
| wait() | 强迫一个线程等待。 |
| notify() | 通知一个线程继续运行。 |
| setPriority() | 设置一个线程的优先级。 |

## Java 中垃圾回收有什么目的？什么时候进行垃圾回收？

1、 垃圾回收是在内存中存在没有引用的对象或超过作用域的对象时进行的。

2、 垃圾回收的目的是识别并且丢弃应用不再使用的对象来释放和重用资源。

## 线程之间如何通信及线程之间如何同步

1、 在并发编程中，我们需要处理两个关键问题：线程之间如何通信及线程之间如何同步。通信是指线程之间以如何来交换信息。一般线程之间的通信机制有两种：共享内存和消息传递。

2、 Java 的并发采用的是共享内存模型，Java 线程之间的通信总是隐式进行，整个通信过程对程序员完全透明。如果编写多线程程序的 Java 程序员不理解隐式进行的线程之间通信的工作机制，很可能会遇到各种奇怪的内存可见性问题。

## Java 内存模型

共享内存模型指的就是 Java 内存模型(简称 JMM)，JMM 决定一个线程对共享变量的写入时,能对另一个线程可见。从抽象的角度来看，JMM 定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/

写共享变量的副本。本地内存是 JMM 的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。

从上图来看，线程 A 与线程 B 之间如要通信的话，必须要经历下面 2 个步骤：

1、 首先，线程 A 把本地内存 A 中更新过的共享变量刷新到主内存中去。

2、 然后，线程 B 到主内存中去读取线程 A 之前已更新过的共享变量。

**下面通过示意图来说明线程之间的通信**

**总结：什么是 Java 内存模型：**

java 内存模型简称 jmm，定义了一个线程对另一个线程可见。共享变量存放在主内存中，每个线程都有自己的本地内存，当多个线程同时访问一个数据的时候，可能本地内存没有及时刷新到主内存，所以就会发生线程安全问题。

## 如果对象的引用被置为 null，垃圾收集器是否会立即释放对象占用的内存？

1、 不会，在下一个垃圾回调周期中，这个对象将是被可回收的。

2、 也就是说并不会立即被垃圾收集器立刻回收，而是在下一次垃圾回收时才会释放其占用的内存。

## finalize()方法什么时候被调用？析构函数(finalization)的目的是什么？

**1、** 垃圾回收器（garbage colector）决定回收某对象时，就会运行该对象的 finalize()方法； finalize 是 Object 类的一个方法，该方法在 Object 类中的声明 protected void finalize() throws Throwable { } 在垃圾回收器执行时会调用被回收对象的 finalize()方法，可以覆盖此方法来实现对其资源的回收。注意：一旦垃圾回收器准备释放对象占用的内存，将首先调用该对象的 finalize()方法，并且下一次垃圾回收动作发生时，才真正回收对象占用的内存空间

**2、** GC 本来就是内存回收了，应用还需要在 finalization 做什么呢？ 答案是大部分时候，什么都不用做(也就是不需要重载)。只有在某些很特殊的情况下，比如你调用了一些 native 的方法(一般是 C 写的)，可以要在 finaliztion 里去调用 C 的释放函数。

**3、** Finalizetion 主要用来释放被对象占用的资源（不是指内存，而是指其他资源，比如文件(File Handle)、端口(ports)、数据库连接(DB Connection)等）。然而，它不能真正有效地工作。

## 什么是重排序

程序执行的顺序按照代码的先后顺序执行。

一般来说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，进行重新排序（重排序），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。

```
int a = 5;//语句 1
int r = 3;//语句 2
a = a + 2;//语句 3
r = a*a;   //语句 4
```

则因为重排序，他还可能执行顺序为（这里标注的是语句的执行顺序） 2-1-3-4，1-3-2-4 但绝不可能 2-1-4-3，因为这打破了依赖关系。

显然重排序对单线程运行是不会有任何问题，但是多线程就不一定了，所以我们在多线程编程时就得考虑这个问题了。

## 重排序实际执行的指令步骤

![87_5.png][87_5.png]

**1、** 编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。

**2、** 指令级并行的重排序。现代处理器采用了指令级并行技术（ILP）来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。

**3、** 内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。

这些重排序对于单线程没问题，但是多线程都可能会导致多线程程序出现内存可见性问题。

## 重排序遵守的规则

**as-if-serial:**

**1、** 不管怎么排序，结果不能改变

2、 不存在数据依赖的可以被编译器和处理器重排序

3、 一个操作依赖两个操作，这两个操作如果不存在依赖可以重排序

4、 单线程根据此规则不会有问题，但是重排序后多线程会有问题

## as-if-serial 规则和 happens-before 规则的区别

1、 as-if-serial 语义保证单线程内程序的执行结果不被改变，happens-before 关系保证正确同步的多线程程序的执行结果不被改变。

2、 as-if-serial 语义给编写单线程程序的程序员创造了一个幻境：单线程程序是按程序的顺序来执行的。happens-before 关系给编写正确同步的多线程程序的程序员创造了一个幻境：正确同步的多线程程序是按 happens-before 指定的顺序来执行的。

3、 as-if-serial 语义和 happens-before 这么做的目的，都是为了在不改变程序执行结果的前提下，尽可能地提高程序执行的并行度。

## 并发关键字 synchronized ？

在 Java 中，synchronized 关键字是用来控制线程同步的，就是在多线程的环境下，控制 synchronized 代码段不被多个线程同时执行。synchronized 可以修饰类、方法、变量。

另外，在 Java 早期版本中，synchronized 属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核

态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对 synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6 对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

## 说说自己是怎么使用 synchronized 关键字，在项目中用到了吗

**synchronized 关键字最主要的三种使用方式：**

**1、** 修饰实例方法：作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁

**2、** 修饰静态方法：也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（ static 表明这是该类的一个静态资源，不管 new 了多少个对象，只有一份）。所以如果一个线程 A 调用一个实例对象的非静态 synchronized 方法，而线程 B 需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。

**3、** 修饰代码块：指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

**总结：**

synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是是给 Class 类上锁。synchronized 关键字加到实例方法上是给对象实例上锁。

尽量不要使用 synchronized(String a) 因为 JVM 中，字符串常量池具有缓存功能！

## 单例模式了解吗？给我解释一下双重检验锁方式实现单例模式！”

**双重校验锁实现对象单例（线程安全）**

**说明：**

双锁机制的出现是为了解决前面同步问题和性能问题，看下面的代码，简单分析下确实是解决了多线程并行进来不会出现重复 new 对象，而且也实现了懒加载

```
public class Singleton {
        private volatile static Singleton uniqueInstance;

        private Singleton() {
        }

       public static Singleton getUniqueInstance() {
              //先判断对象是否已经实例过，没有实例化过才进入加锁代码
              if (uniqueInstance == null) {
                  //类对象加锁
                  synchronized (Singleton.class) {
                      if (uniqueInstance == null) {
                          uniqueInstance = new Singleton();
                      }
                  }
              }
          return uniqueInstance;
      }
```

```
}
```

另外，需要注意 uniqueInstance 采用 volatile 关键字修饰也是很有必要。

uniqueInstance 采用 volatile 关键字修饰也是很有必要的， uniqueInstance = new Singleton(); 这段代码其实是分为三步执行：

1、 为 uniqueInstance 分配内存空间

2、 初始化 uniqueInstance

3、 将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。

使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

## 说一下 synchronized 底层实现原理?

1、 Synchronized 的语义底层是通过一个 monitor（监视器锁）的对象来完成，

2、 每个对象有一个监视器锁(monitor)。每个 Synchronized 修饰过的代码当它的 monitor 被占用时就会处于锁定状态并且尝试获取 monitor 的所有权 ，过程：

1、 如果 monitor 的进入数为 0，则该线程进入 monitor，然后将进入数设置为 1，该线程即为 monitor 的所有者。

**2、** 如果线程已经占有该 monitor，只是重新进入，则进入 monitor 的进入数加 1.

**3、** 如果其他线程已经占用了 monitor，则该线程进入阻塞状态，直到 monitor 的进入数为 0，再重新尝试获取 monitor 的所有权。

synchronized 是可以通过 反汇编指令 javap 命令，查看相应的字节码文件。

## synchronized 可重入的原理

重入锁是指一个线程获取到该锁之后，该线程可以继续获得该锁。底层原理维护一个计数器，当线程获取该锁时，计数器加一，再次获得该锁时继续加一，释放锁时，计数器减一，当计数器值为 0 时，表明该锁未被任何线程所持有，其它线程可以竞争获取锁。

## 什么是自旋

很多 synchronized 里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然 synchronized 里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在 synchronized 的边界做忙循环，这就是自旋。如果做了多次循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

忙循环：就是程序员用循环让一个线程等待，不像传统方法 wait(), sleep() 或 yield() 它们都放弃了 CPU 控制，而忙循环不会放弃 CPU，它就是在运行一个空循环。这么做的目的是为了保留 CPU 缓存，在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了。

## 多线程中 synchronized 锁升级的原理是什么?

synchronized 锁升级原理：在锁对象的对象头里面有一个 threadid 字段，在第一次访问的时候 threadid 为空，jvm 让其持有偏向锁，并将 threadid 设置为其线程 id，再次进入的时候会先判断 threadid 是否与其线程 id 一致，如果一致则可以直接使用此对象，如果不一致，则升级偏向锁为轻量级锁，通过自旋循环一定次数来获取锁，执行一定次数之后，如果还没有正常获取到要使用的对象，此时就会把锁从轻量级升级为重量级锁，此过程就构成了 synchronized 锁的升级。

**锁的升级的目的：**

锁升级是为了减低了锁带来的性能消耗。在 Java 6 之后优化 synchronized 的实现方式，使用了偏向锁升级为轻量级锁再升级到重量级锁的方式，从而减低了锁带来的性能消耗。

**1、** 偏向锁，顾名思义，它会偏向于第一个访问锁的线程，如果在运行过程中，同步锁只有一个线程访问，不存在多线程争用的情况，则线程是不需要触发同步的，减少加锁 / 解锁的一些 CAS 操作（比如等待队列的一些 CAS 操作），这种情况下，就会给线程加一个偏向锁。 如果在运行过程中，遇到了其他线程抢占锁，则持有偏向锁的线程会被挂起，JVM 会消除它身上的偏向锁，将锁恢复到标准的轻量级锁。

**2、** 轻量级锁是由偏向所升级来的，偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁争用的时候，轻量级锁就会升级为重量级锁；

**3、** 重量级锁是 synchronized ，是 Java 虚拟机中最为基础的锁实现。在这种状态下，Java 虚拟机会阻塞加锁失败的线程，并且在目标锁被释放的时候，唤醒这些线程。