Java 多线程

多线程基础（重点）

1. 线程和进程的区别

进程：指系统正在运行的一个应用程序；程序一旦运行就是进程；进程是系统资源分配的最小单位。

线程：**线程是操作系统调度的最小单元。**进程是系统执行的最小单位。一个进程在其执行的过程中可以产生多个线程。

线程是进程的一个实体，比进程更小的独立运行基本单位。

一个程序下至少有一个进程，一个进程下至少有一个线程，一个进程下也可以有多个线程来增加程序的执行速度。

进程和进程之间的资源无法共享，线程和线程之间的资源可以共享。

2. 并行和并发有什么区别?

并行：同一时刻，多个任务同时执行；多个处理器或多核处理器同时处理多个任务

并发：同一时刻，多个任务交替执行；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。

3. 用户线程与守护线程是什么?

1. 用户线程: 也叫工作线程，运行在前台，执行具体的任务。

2. 守护线程: 在后台运行，一般是为工作线程服务的，当所有的用户线程结束，守护线程自动结束

3. 常见的守护线程: 垃圾回收机制

4. 创建线程有哪几种方式?

创建线程有三种方式：

（1）继承 Thread 类， 重写 run 方法；

（2）实现 Runnable 接口，重写 run 方法；

（3）实现 Callable 接口，需要返回值类型。

5. 线程的 run() 和 start() 有什么区别?

调用 start() 方法，会启动一个线程并使线程进入了就绪状态，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，所以这不是多线程工作。

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。

总结： 调用 `start()` 方法方可启动线程并使线程进入就绪状态，直接执行 `run()` 方法的话不会以多线程的方式执行。

6. 线程的状态

（1）NEW 新建状态，线程被创建出来但没有被调用 `start()`。

（2）RUNNABLE 就绪(可运行状态)，线程被调用了 `start()`，等待运行的状态。

（3）running :运行状态，就绪状态的线程被 cpu 调度后，执行程序代码

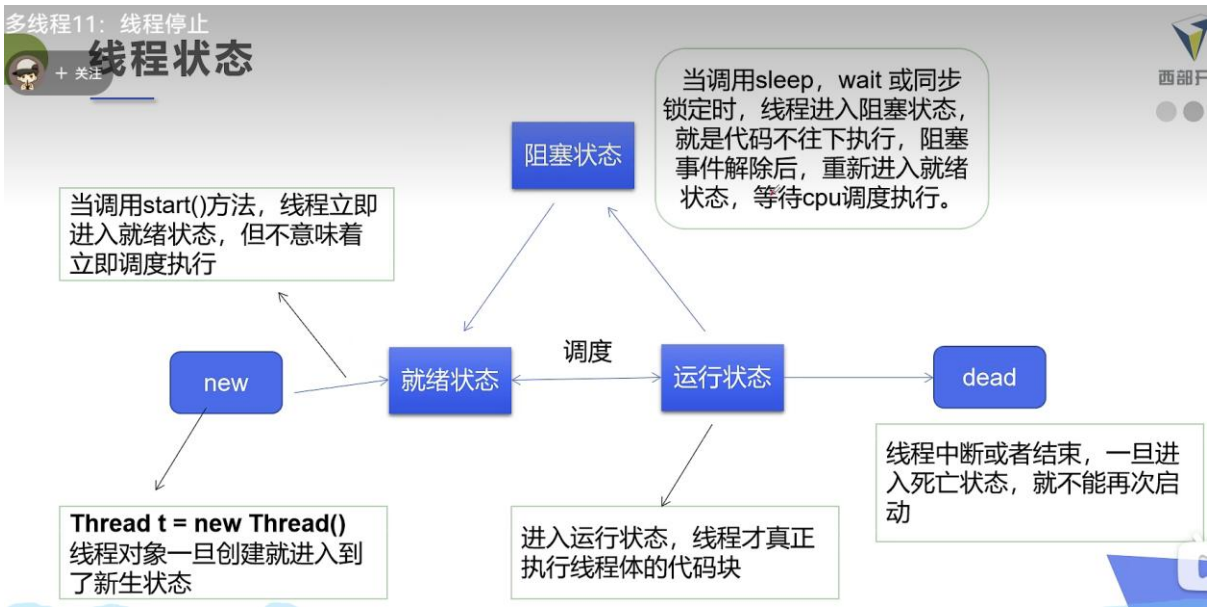
（4）BLOCKED 阻塞状态，需要等待锁释放。

（5）WAITING 持续等待状态，表示该线程需要等待其他线程做出一些特定动作（通知或中断）。

如通过 wait()方法进行等待的线程等待一个 notify()或者 notifyAll()方法，通过 join()方法进行等待的线程等待目标线程运行结束而唤醒，

（6）TIMED_WAITING 超时等待状态，可以在指定的时间后自行返回而不是像 WAITING 那样一直等待。如 sleep(3000)方法。

（7）TERMINATED 终止状态，表示该线程已经运行完毕。



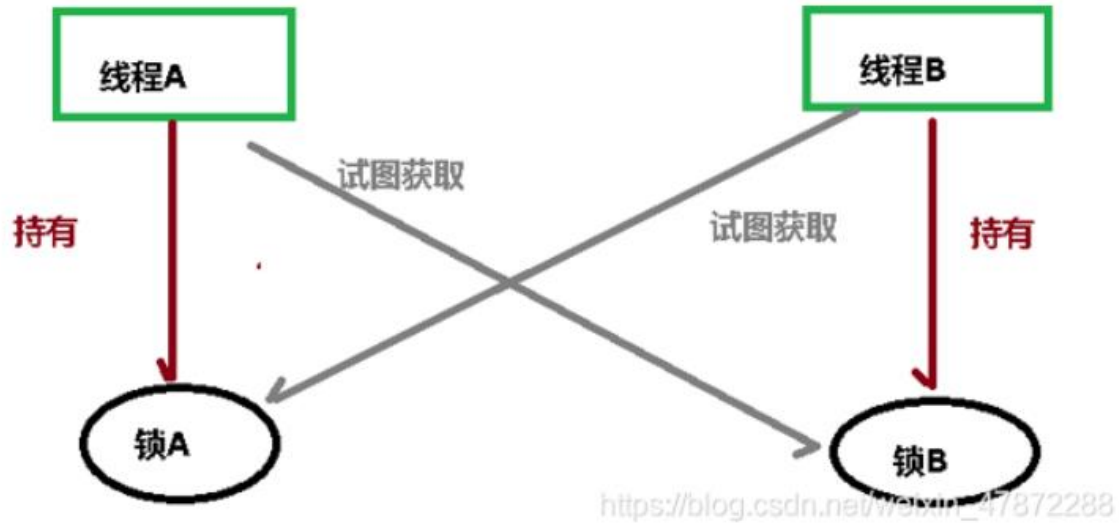
7. 什么上下文切换

线程在执行过程中会有自己的运行条件和状态（也称上下文），线程切换就是需要保存当前线程的上下文，等线程下次占用 CPU 的时候恢复，并加载下一个将要占用 CPU 的线程的上下文。（cpu 控制权从一个正在运行的线程切换到另一个就绪并等待获取 cpu 执行权的线程，但切换之前会保存自己的状态，等下次再切换时，可以再加载这个任务的状态）

8. 什么是死锁?

两个或两个以上的进程在执行过程中，因争夺资源而造成一种互相等待的现象称为死锁，若无外力干涉，它们都无法再执行下去。

举例：线程 A 持有锁 A，试图获取锁 B，线程 B 持有锁 B，试图获取锁 A，就会发生 AB 两个线程由于互相持有对方需要的锁，而发生的阻塞现象，我们称为死锁。



9. 产生死锁的条件：

（1）线程互斥：该资源任意一个时刻只由一个线程占用。

（2）不可抢占：线程已获得的资源在未使用完之前不能被其他线程强行剥夺，只有自己使用完毕后才释放资源。

（3）保持请求：一个线程因请求资源而阻塞时，对已获得的资源保持不放。

（4）循环等待：若干线程之间形成一种头尾相接的循环等待资源关系。

10. 如何预防和避免线程死锁?

（1）破坏保持请求条件 ： 一次性申请所有的资源。

（2）破坏不可抢占条件 ： 占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。

（3）破坏循环等待条件 ： 靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放。破坏循环等待条件。

11. sleep() 和 wait() 有什么区别?

共同点：两者都可以暂停线程的执行

不同点：

（1）类的不同：sleep()是 Thread 的静态方法，wait() 是 Object 的方法，任何对象实例都能调用

（2）是否释放锁：sleep() 不释放锁；wait() 会释放锁。

（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：sleep() 方法被调用后，线程会自动苏醒；wait()方法，线程不会自动苏醒，需要别的线程使用 notify()/notifyAll()直接唤醒。或者 wait(longtimeout)，wait 方法参数设置一个超时时间，超时后线程会自动苏醒。

12. 如何使用 synchronized?

1、修饰实例方法 （锁当前对象实例）

给当前对象实例加锁，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 （锁当前类）

给当前类加锁，会作用于类的所有对象实例 ，进入同步代码前要获得 **当前 class 的锁**。这是因为静态成员不属于任何一个实例对象，归整个类所有，不属于类的特定实例，被类的所有实例共享。

3、修饰代码块 （锁指定对象/类）

- synchronized(object) 表示进入同步代码库前要获得 **给定对象的锁。**
- synchronized(类.class) 表示进入同步代码前要获得 **给定 Class 的锁**

13. synchronized 和 volatile 有什么区别?

synchronized 关键字和 volatile 关键字是两个互补的存在，而不是对立的存在!

（1）volatile 关键字是线程同步的轻量级实现，所以 volatile 性能肯定比 synchronized 关键字要好 。但是 volatile 关键字只能用于变量，而 synchronized 关键字可以修饰方法以及代码块 。

（2）volatile 关键字不能保证数据的原子性。synchronized 关键字可以保证。

（3）volatile 关键字主要用于解决变量在多个线程之间的可见性，而 synchronized 关键字解决的是多个线程之间访问资源的同步性。

14. synchronized 和 ReentrantLock 有什么区别?

共同点：两者都是可重入锁

重入锁：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果是不可重入锁的话，就会造成死锁。底层原理维护一个计数器，同一个线程每次获取锁，锁的计数器都自增 1，所以要等到锁的计数器下降为 0 时才能释放锁。

不同点：

1. sychronized 是一个关键字，ReentrantLock 是一个类
2. sychronized 会自动的加锁与释放锁，ReentrantLock 需要程序员手动加锁与释放锁
3. sychronized 的底层是 JVM 层面的锁，ReentrantLock 是 API 层面的锁
4. sychronized 是非公平锁，ReentrantLock 可以选择公平锁或非公平锁
5. sychronized 锁的是对象，锁信息保存在对象头中，ReentrantLock 通过代码中 int 类型的 state 标识来标识锁的状态

15. ThreadLocal

让每个线程绑定自己的值，实现每一个线程都有自己的专属本地变量。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。

1. ThreadLocal 是 Java 中所提供的线程本地存储机制，可以利用该机制将数据缓存在某个线程内部，该线程可以在任意时刻、任意方法中获取缓存的数据

2. ThreadLocal 底层是通过 ThreadLocalMap 来实现的，每个 Thread 对象（注意不是 ThreadLocal 对象）中都存在一个 ThreadLocalMap，Map 的 key 为 ThreadLocal 对象，Map 的 value 为需要缓存的值

3. 如果在线程池中使用 ThreadLocal 会造成内存泄漏，因为当 ThreadLocal 对象使用完之后，应该要把设置的 key，value，也就是 Entry 对象进行回收，但线程池中的线程不会回收，而线程对象是通过强引用指向 ThreadLocalMap，ThreadLocalMap 也是通过强引用指向 Entry 对象，线程不被回收，Entry 对象也就不会被回收，从而出现内存泄漏，解决办法是，在使用了 ThreadLocal 对象之后，手动调用 ThreadLocal 的 remove 方法，手动清除 Entry 对象

4. ThreadLocal 经典的应用场景就是连接管理（一个线程持有一个连接，该连接对象可以在不同的方法之间进行传递，线程之间不共享同一个连接）

16. 公平锁和非公平锁

- **公平锁**：锁被释放之后，先申请的线程/进程先得到锁。
- **非公平锁**：锁被释放之后，后申请的线程/进程可能会先获取到锁，是随机或者按照其他优先级排序的。

17. Synchronized 和 Lock 的区别

（1）synchronized 是一个关键字，底层是 JVM 层面的锁；lock 是一个接口，是 API 层面的锁

（2）synchronized 是非公平锁，lock 可以选择公平锁或非公平锁

（3）sychronized 会自动的加锁与释放锁，ReentrantLock 需要程序员手动加锁与释放锁

（4）sychronized 获取锁的时候，假设 A 线程获得锁，B 线程等待，若 A 阻塞，B 会一直等待；而 lock 在这种情况下 B 会尝试去获取锁

（5）lock 可以设置超时时间，支持响应中断；sychronized 不行，竞争不到锁会一直阻塞，容易造成死锁

多线程锁

1. ReentrantLock 有哪些方法

lock(); //获得锁

lockInterruptibly（）//获得锁，但是优先响应中断。

boolean tryLock() //尝试获得锁，成功返回 true

tryLock（long time,TimeUnit unit）//在给定时间内尝试获得锁。

unlock(); //释放锁

2. ReentrantLock 中的公平锁和非公平锁的底层实现

首先不管是公平锁和非公平锁，它们的底层实现都会使用 AQS 来进行排队，它们的区别在于：线程在使用 lock()方法加锁时，如果是公平锁，会先检查 AQS 队列中是否存在线程在排队，如果有线程在排队，则当前线程也进行排队，如果是非公平锁，则不会去检查是否有线程在排队，而是直接竞争锁。

不管是公平锁还是非公平锁，一旦没竞争到锁，都会进行排队，当锁释放时，都是唤醒排在最前面的线程，所以非公平锁只是体现在了线程加锁阶段，而没有体现在线程被唤醒阶段。

另外，ReentrantLock 是可重入锁，不管是公平锁还是非公平锁都是可重入的。

AQS

AbstractQueuedSynchronizer，抽象队列同步器，一个抽象类，主要用来构建锁和同步器。

原理：通过内置的 CLH(FIFO)队列的变种——一个虚拟的双向队列，来完成资源获取，线程的排队工作，将每条将要去抢占资源的线程封装成一个 Node 节点来实现锁的分配，有一个 int 类型的变量 state 表示持有锁的状态，通过 CAS 完成对 state 值的修改

CAS-(compare and swap),比较并交换

```java
// 共享变量，使用 volatile 修饰保证线程可见性
private volatile int state;
```

Java

AQS 是 JUC 内容中最重要的基石，主要用于解决锁分配给谁的问题

JUC 常见的同步工具类（了解）

CountDownLatch 有什么用？（减少计数）

`CountDownLatch` 允许 `count` 个线程阻塞在一个地方，直至所有线程的任务都执行完毕。

举例场景：

我们要读取处理 6 个文件，这 6 个任务都是没有执行顺序依赖的任务，但是我们需要返回给用户的时候将这几个文件的处理的结果进行统计整理。

CountDownLatch 表示计数器，可以给 CountDownLatch 设置一个数字，一个线程调用 CountDownLatch 的 await()将会阻塞，其他线程可以调用 CountDownLatch 的 countDown()方法来对 CountDownLatch 中的数字减一，当数字被减成 0 后，所有

await 的线程都将被唤醒。对应的底层原理就是，调用 await()方法的线程会利用 AQS 排队，一旦数字被减为 O，则会将 AQS 中排队的线程依次唤醒。

CyclicBarrier 有什么用？（可循环的屏障）循环栅栏

CyclicBarrier 内部通过一个 count 变量作为计数器，count 的初始值为 parties 属性的初始化值，每当一个线程到了栅栏（屏障）这里了，调用 await()方法，那么就将计数器减 1。如果 count 值为 0 了，表示这是最后一个线程到达栅栏，栅栏才会打开，线程才得以通过执行。

Semaphore 有什么用？（信号量）

Semaphore 表示信号量，可以设置许可的个数，表示同时允许最多多少个线程使用该信号量，通过 acquire()来获取许可，如果没有许可可用则线程阻塞，并通过 AQS 来排队，可以通过 release()方法来释放许可，当某个线程释放了某个许可后，会从 AQS 中正在排队的第一个线程开始依次唤醒，直到没有空闲许可。

JMM（Java 内存模型)

1. 什么是 JMM

JMM(Java 内存模型 Java Memory Model，简称 JMM)，是一种抽象的概念，描述的是一组规范，通过这组规范，定义了程序中(尤其是多线程)各个变量的读写访问方式，并决定一个线程对共享变量的写入时，能对另一个线程可见。JMM 的关键技术点都是围绕**多线程的原子性、可见性和有序性**展开的。

能干嘛?

（1）通过 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。

（2）屏蔽各种硬件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。

2. Java 内存区域和 JMM 有何区别?

（1）JVM 内存结构和 Java 虚拟机的运行时区域相关，定义了 JVM 在运行时如何分区存储程序数据，就比如说堆主要用于存放对象实例。

（2）Java 内存模型和 Java 的并发编程相关，抽象了线程和主内存之间的关系，就比如说线程之间的共享变量必须存储在主内存中，规定了从 Java 源代码到 CPU 可执行指令的这个转化过程要遵守哪些和并发相关的原则和规范，其主要目的是为了简化多线程编程，增强程序可移植性的。

3. 并发编程三个重要特性

（1）原子性：指一次操作或者多次操作是不可中断的，即多线程环境下，操作不能被其他线程干扰，要么全部都执行，要么都不执行

（2）可见性：当一个线程对共享变量进行了修改，另外的线程可以立刻看到（synchronized 、volatile）

（3）有序性：程序执行的顺序按照代码的先后顺序执行。但是由于**指令重排序**问题，代码的执行顺序未必就是编写代码时候的顺序。

重排序：为了提供性能，编译器和处理器通常会对指令序列进行**重新排序。**Java 规范规定 JVM 线程内部保证**串行化语义**，即只要程序的最终结果与它顺序化执行的结果**相等**，那么指令的执行顺序可以与代码顺序**不一致**，此过程叫指令的**重排序。**

指令重排序可以保证串行语义一致，但是没有义务保证多线程间的语义也一致，所以在多线程下，指令重排序可能会导致一些问题。

在 java 中，**volatile** 关键字可以禁止指令进行重排序优化

volatile 关键字

volatile 是 Java 虚拟机提供的轻量级的同步机制

volatile 关键字能保证数据的可见性，但不能保证数据的原子性

synchronized 关键字两者都能保证。

1. volatile 可以保证变量的可见性

保证不同线程对某个变量完成操作后结果及时可见，即该共享变量一旦改变所有线程立即可见。被 volatile 修改的变量有以下特点：

1. 线程中**读取**的时候，每次读取都会去**主内存中**读取共享变量最新的值 ，然后将其复制到工作内存
2. 线程中修改了工作内存中变量的副本，修改之后会**立即刷新到主内存**

2. volatile 不能保证原子性

volatile 关键字能保证变量的可见性，不能保证对变量的操作是原子性的。

JVM 只是保证从主内存加载到线程工作内存的值是最新的，也仅是数据加载时是最新的。但是在多线程环境下，对于一个变量，如果第二个线程在第一个线程读取旧值和写回新值的期间，也读取操作了这个变量，就会出现写丢失问题。

举例子：比如说，对于一个共享变量，两个线程在计算的时候，一个线程计算后还没提交，第二个线程已经计算后提交了，那么第一个线程的操作计算将会作废去读主内存最新的值，操作出现写丢失问题。

3. 底层：内存屏障

内存屏障其实就是一种 jvm 指令，java 内存模型的重排规则会要求 Java 编译器在生成 JVM 指令时插入特定的内存屏障指令 ，通过这些内存屏障指令，volatile 实现了 Java 内存模型中的可见性和有序性，*但 volatile 无法保证原子性 。*

内存屏障之前**的所有**写**操作都要回写到主内存，**内存屏障之后**的所有**读**操作都能获得内存屏障之前的所有写操作的最新结果(实现了可见性)。**

写屏障(Store Memory Barrier)：告诉处理器在写屏障之前将所有存储在缓存(store bufferes) 中的数据刷新到主内存中。也就是说当看到 Store 屏障指令， 就必须把该指令之前所有写入指令执行完毕才能继续往下执行。

读屏障(Load Memory Barrier)：在读指令之前插入读屏障，让工作内存或 CPU 高速缓存当中的缓存数据失效，重新回到主内存中获取最新数据。

4. 双重检验锁实现对象单例（线程安全）：

什么是双重检验锁?

在实现单例模式时，多线程下可能会引起线程不安全的问题，

第一种解决方法是：使用 synchronized 关键字加锁，但是性能消耗比较大。

第二种解决方法，可以使用双重检验，synchronized 配合 volatile 使用：先判断对象是否已经被初始化，再决定要不要加锁。**初始化的时候会出现加锁，后续的所有调用都不会加锁而直接返回**，同时也可以解决指令重排引起的问题。

```
public class SafeDoubleCheckSingleton
{   //-----这里没加 volatile，实现线程安全的延迟初始化
    private static volatile SafeDoubleCheckSingleton singleton;
    //私有化构造方法
    private SafeDoubleCheckSingleton(){
    }
    //双重锁设计
```

```
    public static SafeDoubleCheckSingleton getInstance(){
        if (singleton == null){
            //1.多线程并发创建对象时，会通过加锁保证只有一个线程能创建对
象
            synchronized (SafeDoubleCheckSingleton.class){
                if (singleton == null){
                    //隐患：多线程环境下，由于重排序，该对象可能还未完成
初始化就被其他线程读取
                    //由于指令重排，会将第二步和第三步进行重排序，某个线
程可能会获得一个未完全初始化的实例
                    singleton = new SafeDoubleCheckSingleton();
                    //实例化分为三步
                    //1.分配对象的内存空间
                    //2.初始化对象
                    //3.设置对象指向分配的内存地址
                }
            }
        }
        //2.对象创建完毕，执行 getInstance()将不需要获取锁，直接返回创建
对象
        return singleton;
    }
}
```

Java

Java 常见并发容器总结

1. CopyOnWriteArrayList

线程安全的 List，在读多写少的场合性能非常好，远远好于 Vector。用到**写时复制技术**

1. 首先 CopyOnWriteArrayList 内部也是用数组来实现的，在向 CopyOnWriteArrayList 添加元素时，会复制一个新的数组，写操作在新数组上进行，读操作在原数组上进行

2. 并且，写操作会加锁，防止出现并发写入丢失数据的问题，保证了同步，避免了多线程写的时候会 copy 出多个副本出来。

3. 写操作结束之后会把原数组指向新数组

4. CopyOnWriteArrayLlist 允许在写操作时来读取数据，大大提高了读的性能，因此适合读多写少的应用场景，但是 CopyOnWriteArraylist 会比较占内存，同时可能读到的数据不是实时最新的数据，所以不适合实时性要求很高的场景

2. CopyOnWriteArraySet

底层是 CopyOnWriteArrayList

3. ConcurrentHashMap

HashMap 不是线程安全的，HasTable 的内部方法都被 synchronized 修饰了，所以是线程安全的。

ConcurrentHashMap 内部使用粒度更细的分段锁机制，分段锁简单来说就是将数据进行分段，每一段锁用于锁容器中的一部分数据，那么当多线程访问容器里的不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效地提高并发访问效率。

4. BlockingQueue

阻塞队列（BlockingQueue）被广泛使用在“生产者-消费者”问题中，其原因是BlockingQueue 提供了可阻塞的插入和移除的方法。当队列容器已满，生产者线程会被阻塞，直到队列未满；当队列容器为空时，消费者线程会被阻塞，直至队列非空时为止。

线程池（重点）

1. 为什么要用线程池?

池化技术的思想主要是为了减少每次获取资源的消耗，提高对资源的利用率。线程池的好处：

- **降低资源消耗。**通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
- **提高响应速度。**当任务到达时，任务可以不需要等到线程创建就能立即执行。
- **提高线程的可管理性。**线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

2. ThreadPoolExecutor 参数（线程池参数）

```
/**
     * 用给定的初始参数创建一个新的 ThreadPoolExecutor。
     */
    public ThreadPoolExecutor(int corePoolSize,//线程池的核心线程数量
                              int maximumPoolSize,//线程池的最大线程数
                              long keepAliveTime,//当线程数大于核心线程数时，多余的空闲线程存活的最长时间
                              TimeUnit unit,//时间单位
                              BlockingQueue<Runnable> workQueue,//任务队列，用来储存等待执行任务的队列
```

```
                                ThreadFactory threadFactory,//线程工
厂，用来创建线程，一般默认即可
                                RejectedExecutionHandler handler//拒绝
(饱和)策略，当提交的任务过多而不能及时处理时，我们可以定制策略来处理任务
                                ) {
        if (corePoolSize < 0 ||
            maximumPoolSize <= 0 ||
            maximumPoolSize < corePoolSize ||
            keepAliveTime < 0)
            throw new IllegalArgumentException();
        if (workQueue == null || threadFactory == null || handler ==
null)
            throw new NullPointerException();
        this.corePoolSize = corePoolSize;
        this.maximumPoolSize = maximumPoolSize;
        this.workQueue = workQueue;
        this.keepAliveTime = unit.toNanos(keepAliveTime);
        this.threadFactory = threadFactory;
        this.handler = handler;
    }
```

Java

3. 线程池的实现原理

执行 execute 方法，创建新的线程，分四种情况

（1）如果当前运行的线程少于 corePoolSize，则创建新线程来执行任务（注意，执行这一步骤需要获取全局锁）。

（2）如果运行的线程等于或多于 corePoolSize，则将任务加入 BlockingQueue。

（3）如果无法将任务加入 BlockingQueue（队列已满），则创建新的线程来处理任务）。

（4）如果创建新线程将使当前运行的线程超出 maximumPoolSize，任务将被拒绝，并调用 RejectedExecutionHandler.rejectedExecution()方法。

4. 线程池中 submit() 和 execute() 方法有什么区别?

execute()：只能执行 Runnable 类型的任务，没有返回值

submit()：可以执行 Runnable 和 Callable 类型的任务。

Callable 类型的任务可以获取执行的返回值，而 Runnable 执行无返回值。

5. ThreadPoolExecutor 饱和策略有哪些?

- AbortPolicy：直接抛出异常。
- CallerRunsPolicy：只用调用者所在线程来运行任务。
- DiscardOldestPolicy：丢弃队列里最近的一个任务，并执行当前任务。
- DiscardPolicy：不处理，丢弃掉。

其它

同步容器，并发容器

同步容器：并发访问 ArrayList、LinkedList、HashMap 等容器时，会出现问题。所以 java 提供同步容器供用户使用，主要包括： Vector 、 HashTable 、 Collections.synchronized 方法。可以简单地理解为通过 synchronized 来实现同步的容器，同步容器会导致多个线程中对容器方法调用的串行执行，降低并发性，效率低。

并发容器：并发容器是针对多个线程并发访问而设计的，在 jdk5.0 引入了 concurrent 包，其中提供了很多并发容器，如 ConcurrentHashMap、CopyOnWriteArrayList 等。

- **CopyOnWrite 容器**：、CopyOnWriteArrayList、CopyOnWriteArraySet

CopyOnWriteArrayList：写时复制技术，写时复制一份新的，在新的上面修改，然后把引用指向新的，适合读多写少的场合。由于写操作的时候会进行复制，会消耗内存。

CopyOnWriteArraySet，java.util.ArrayList 的线程安全版本，基于 CopyOnWriteArrayList

- **CocurrentMap 的实现类**：ConcurrentHashMap、ConcurrentSkipListMap

  HashMap 不是线程安全的，HasTable 的内部方法都被 synchronized 修饰了，所以是线程安全的。ConcurrentHashMap 内部使用粒度更细的分段锁机制，分段锁简单来说就是将数据进行分段，每一段锁用于锁容器中的一部分数据，那么当多线程访问容器里的不同数据段的数据时，线程间就不会存在锁竞争，从而可以有效地提高并发访问效率。

  ConcurrentSkipListMap：使用 SkipList(跳表)实现排序，Skip list（跳表）是一种可以代替平衡树的数据结构，默认是按照 Key 值升序的。Skip list 让已排序的数据分布在多层链表中，通过"空间来换取时间"的一个算法。