第4章 Java并发编程面试题汇总

考生应该如何回答

4.1 假如只有一个cpu，单核，多线程还有用吗？

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门1》

这道题想考察什么？

是否了解并发相关的理论知识

考察的知识点

1. cpu多线程的基本概念
2. 操作系统的调度任务机制
3. CPU密集型和IO密集型理论

考生应该如何回答

CPU的执行速度要远大于IO的过程，因此在大多数情况下增加一些复杂的CPU计算都比增加一次IO要快。单核CPU可以通过给每个线程分配CPU时间片（时间单元）来实现多线程机制。由于CPU频率很高，故时间单元非常短。所以单核也可以实现多线程机制。

从用户体现上说，单核多线程也能够减少用户响应时间，例如web页面，也是防止IO阻塞。处理器的数量和并不影响程序结构, 所以不管处理器的个数多少, 程序都可以通过使用多线程得到简化。

4.2 synochnized修饰普通方法和静态方法的区别？什么是可见性?（小米）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门1》

这道题想考察什么？

是否了解Java并发编程的相关知识

考察的知识点

1. sychronied的原理
2. 并发的特性

考生应该如何回答

sychronied是Java中并发编程的重要关键字之一。在并发编程中synchronized一直是解决线程安全问题，它可以保证原子性，可见性，以及有序性。

- 原子性：原子是构成物质的基本单位，所以原子的意思代表着一“不可分”。由不可分可知，具有原子性的操作也就是拒绝线程调度器中断。

- 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，称为可见性。
- 有序性：程序按照代码的先后顺序执行。编译器为了优化性能，有时会改变程序中语句的顺序，但是不会影响最终的结果。有序性经典的例子就是利用DCL双重检查创建单例对象。

synchronized可以修饰方法，也能够使用`synchronized(obj){}`定义同步代码块。

- 修饰方法：
    1. 实例方法，作用于当前实例加锁，进入方法前需要获取当前实例的锁;
    2. 静态方法，作用于当前类对象加锁，进入方法前需要获取当前类对象的锁;
- 修饰代码块，指定加锁对象，对给定对象加锁，进入代码块前要获得给定对象的锁。

使用sychronied修饰普通方法和静态方法，其实也等价于`synchronized(this){}`与`synchronized(class){}`。

4.3 Synchronized在JDK1.6之后做了哪些优化（京东）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门1》

这道题想考察什么？

对并发编程的掌握

考察的知识点

并发与synchronized原理

考生如何回答

synchronized是Java中非常重要的一个关键字，对于Android开发同学来说，考虑到多线程的情况，一般都直接使用到synchronized关键字对方法或者对象上锁。但是问题是为什么加上synchronized关键字就能实现锁，它的原理是怎么回事呢？

字节码

如果我们使用javap -v xxx.class 反编译这样一个class文件

```
public static void main(String[] args) {
        synchronized (InjectTest.class) {
            System.out.println("hello!");
        }
}
```

此时我们获得到结果为：

```
public static void main(java.lang.String[]) throws java.lang.InterruptedException;
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=3, args_size=1
       0: ldc           #2                  // class com/enjoy/asminject/InjectTest
       2: dup
       3: astore_1
       4: monitorenter
       5: getstatic     #3                  // Field java/lang/System.out:Ljava/io/PrintStream;
       8: ldc           #4                  // String hello!
      10: invokevirtual #5                  // Method java/io/PrintStream.println:(Ljava/lang/String;)V
      13: aload_1
      14: monitorexit
      15: goto          23
      18: astore_2
      19: aload_1
      20: monitorexit
      21: aload_2
      22: athrow
      23: return
```

可以看到javap的输出信息中存在：monitorenter与monitorexit指令。这就代表了同步代码块的入口与出口。

这里的monitor是：对象监视器。在JVM中,每个对象都会和一个对象监视器（monitor）相关联。monitorenter指令插入到同步代码块开始的位置、monitorexit指令插入到同步代码块结束位置，jvm需要保证每个monitorenter都有一个monitorexit对应。这两个指令，本质上都是对 对象监视器(monitor)进行获取，这个过程是排他的，也就是说同一时刻只能有一个线程获取到由synchronized所保护对象的监视器。线程执行到monitorenter指令时，会尝试获取对象所对应的monitor所有权，也就是尝试获取对象的锁；而执行monitorexit，就是释放monitor的所有权。 如果其他线程已经占用了monitor，则当前线程进入阻塞状态。

当然这是jdk1.6之前的行为，而**jdk1.6以后为了减少获得锁和释放锁带来的性能消耗，对synchronized锁进行了优化，包含偏向锁、轻量级锁、重量级锁;**

关于锁类型与相关信息的信息都是存放在锁对象的对象头中 ，在了解偏向锁、轻量级锁、重量级锁之前，我们必须先认识一下什么是对象头！

Java对象头

对象在虚拟机内存中的布局分为三块区域：对象头、实例数据和对齐填充；Java对象头是实现synchronized的锁对象的基础，一般而言，synchronized使用的锁对象是存储在Java对象头里。

对象头由：存储对象自身的运行时数据的Mark Word 32位系统中4 + 指向类的指针 kClass pointer ,如果是数组对象还会有数组长度 Array Length。

其中mark word中就存储了锁状态：

锁状态	25 bit		4 bit	1 bit	2 bit
	23 bit	2 bit		是否偏向锁	锁标志位
无锁	对象的hashCode		分代年龄	0	01
偏向锁	线程ID	Epoch	分代年龄	1	01
轻量级锁	指向栈中锁记录的指针				00
重量级锁	指向重量级锁的指针				10
GC标记	空				11

在无锁状态下，mark word中的数据为：

无锁	对象的hashCode	分代年龄	0	01

包含对象hashcode，gc年龄，是否偏向锁与锁标志信息。

偏向锁

而偏向锁下，数据则为：

偏向锁	线程ID	Epoch	分代年龄	1	01

拥有锁的线程ID, epoch 大家可以先理解为校验位，同时 是否偏向锁标记由相对无锁状态下的0变为1。

首先之所以会引入偏向锁是因为：大多数情况下锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低而引入了偏向锁，减少不必要的操作。

在程序进入同步代码块时，会访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01，若为偏向锁状态，则查看偏向锁状态下线程ID是否指向当前线程。如果是则直接执行同步代码。但是mark word中记录的线程ID如果不是当前线程，则通过CAS比较与交换尝试修改对象头获得锁。CAS操作成功则可以直接执行同步代码，否则表示有其他线程竞争，此时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁 ，然后被阻塞的线程继续往下执行同步代码。

轻量级锁

轻量级锁	指向栈中锁记录的指针	00

轻量级锁状态下，代码进入同步块时，如果同步对象锁状态为无锁状态， 虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，接着虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针 。这个操作如果成功则代表获取到了锁，但是如果失败，则会检查对象Mark Word是不是指向当前线程栈帧中的锁记录，如果是，则说明本身当前线程就拥有此对象的锁，就可以直接执行同步代码。否则说明锁对象被其他线程获取，当前线程是竞争者，那么当前线程会自旋等待锁，也就是不断重试，当重试一定次数后，总不能一直重试下去吧，太耗CPU了。所以这时候就要升级为重量级锁。

重量级锁

重量级锁就是通过对象监视器（monitor）实现，其中monitor的本质是依赖于底层操作系统的Mutex Lock实现，操作系统实现线程之间的切换需要从用户态到内核态的切换，切换成本非常高。主要是，当系统检查到锁是重量级锁之后，会把等待想要获得锁的线程进行**阻塞**，被阻塞的线程不会消耗cpu。但是阻塞或者唤醒一个线程时，都需要操作系统来帮忙，这就需要从**用户态**转换到**内核态**，而转换状态是需要消耗很多时间的，有可能比用户执行代码的时间还要长。

4.4 CAS无锁编程的原理（字节跳动）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《并发基础与CAS基本原理》

这道题想考察什么？

并发相关问题，原子操作

考察的知识点

Java并发编程，乐观锁机制

考生如何回答

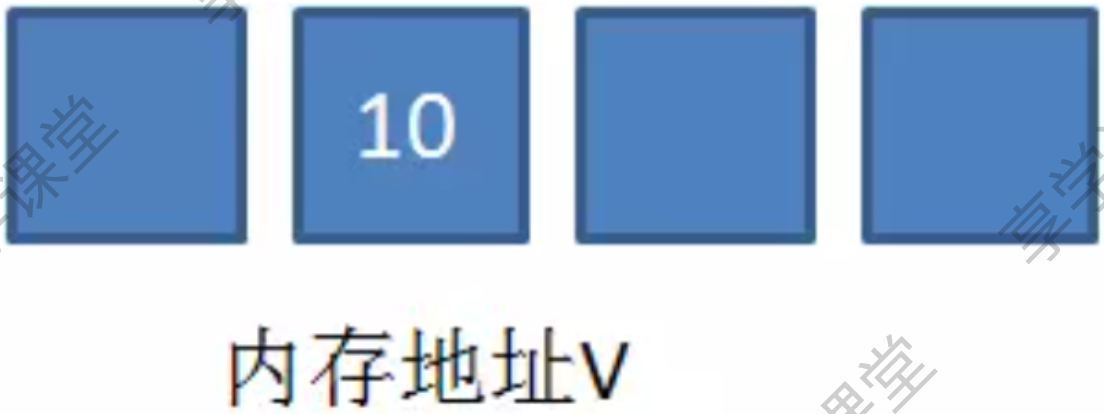
Jdk中java.util.concurrent.atomic包下的类都是采用CAS来实现的。

CAS原理分析

CAS（比较与交换，Compare and swap）是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。实现非阻塞同步的方案称为"无锁编程算法"（ Non-blocking algorithm）。

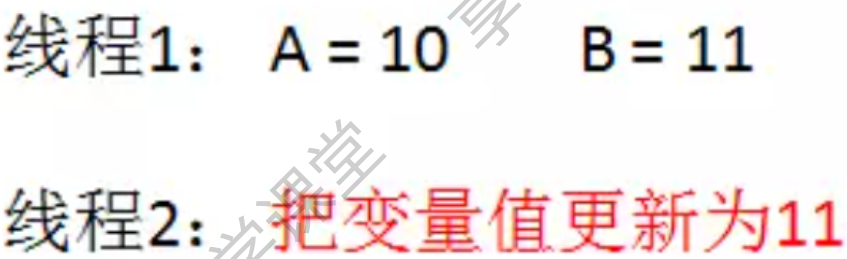
CAS机制当中使用了3个基本操作数：内存地址V，旧的预期值A，要修改的新值B。更新一个变量的时候，只有当变量的预期值A和内存地址V当中的实际值相同时，才会将内存地址V对应的值修改为B。

1.在内存地址V当中，存储着值为10的变量。

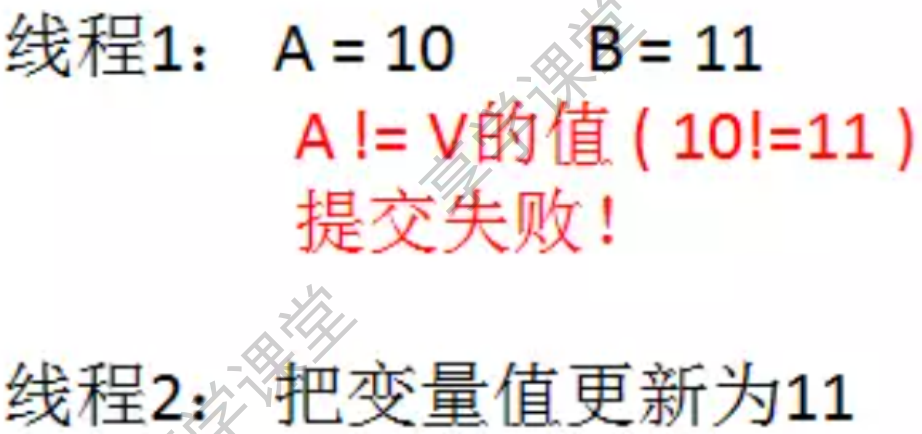


2.此时线程1想要把变量的值增加1。对线程1来说，旧的预期值A=10，要修改的新值B=11。

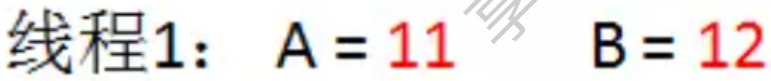
3.在线程1要提交更新之前，另一个线程2抢先一步，把内存地址V中的变量值率先更新成了11。



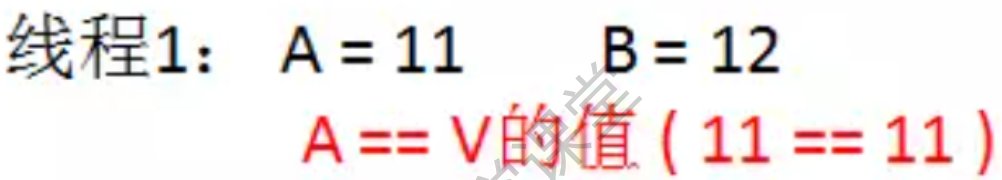
4.线程1开始提交更新，首先进行**A和地址V的实际值比较（Compare）**，发现A不等于V的实际值，提交失败。



5.线程1重新获取内存地址V的当前值，并重新计算想要修改的新值。此时对线程1来说，A=11，B=12。这个重新尝试的过程被称为**自旋**。



6.这一次比较幸运，没有其他线程改变地址V的值。线程1进行**Compare**，发现A和地址V的实际值是相等的。



7.线程1进行**SWAP**，把地址V的值替换为B，也就是12。

内存地址V

线程1： A = 11 B = 12

A == V的值（11 == 11）

地址V的值更新为12

从思想上来说，Synchronized属于**悲观锁**，悲观地认为程序中的并发情况严重，所以严防死守。CAS属于**乐观锁**，乐观地认为程序中的并发情况不那么严重，所以让线程不断去尝试更新。

CAS的缺点

ABA 问题

由于 CAS 设计机制就是获取某两个时刻(初始预期值和当前内存值)变量值，并进行比较更新，所以说如果在获取初始预期值和当前内存值这段时间间隔内，变量值由 A 变为 B 再变为 A，那么对于 CAS 来说是不可感知的，但实际上变量已经发生了变化；解决办法是在每次获取时加版本号，并且每次更新对版本号 +1，这样当发生 ABA 问题时通过版本号可以得知变量被改动过。JDK 1.5 以后的 AtomicStampedReference 类就提供了此种能力，其中的 compareAndSet 方法就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

循环时间长开销大

所谓循环时间长开销大问题就是当 CAS 判定变量被修改了以后则放弃本次修改，但往往为了保证数据正确性该计算会以循环的方式再次发起 CAS，如果多次 CAS 判定失败，则会产生大量的时间消耗和性能浪费；如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

只能保证一个共享变量的原子操作

1. CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效；从 JDK 1.5开始提供了 AtomicReference 类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作

2. Unsafe是CAS的核心类，Java无法直接访问底层操作系统，而是通过本地（native）方法来访问。不过尽管如此，JVM还是开了一个后门，JDK中有一个类Unsafe，它提供了硬件级别的原子操作。
3. valueOffset表示的是变量值在内存中的偏移地址，因为Unsafe就是根据内存偏移地址获取数据的原值的。
4. value是用volatile修饰的，保证了多线程之间看到的value值是同一份。

4.5 AQS原理（小米 京东）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《深入理解并发编程-AQS与JMM》

这道题想考察什么？

是否了解Java并发编程的相关知识？

考察的知识点

AQS的原理

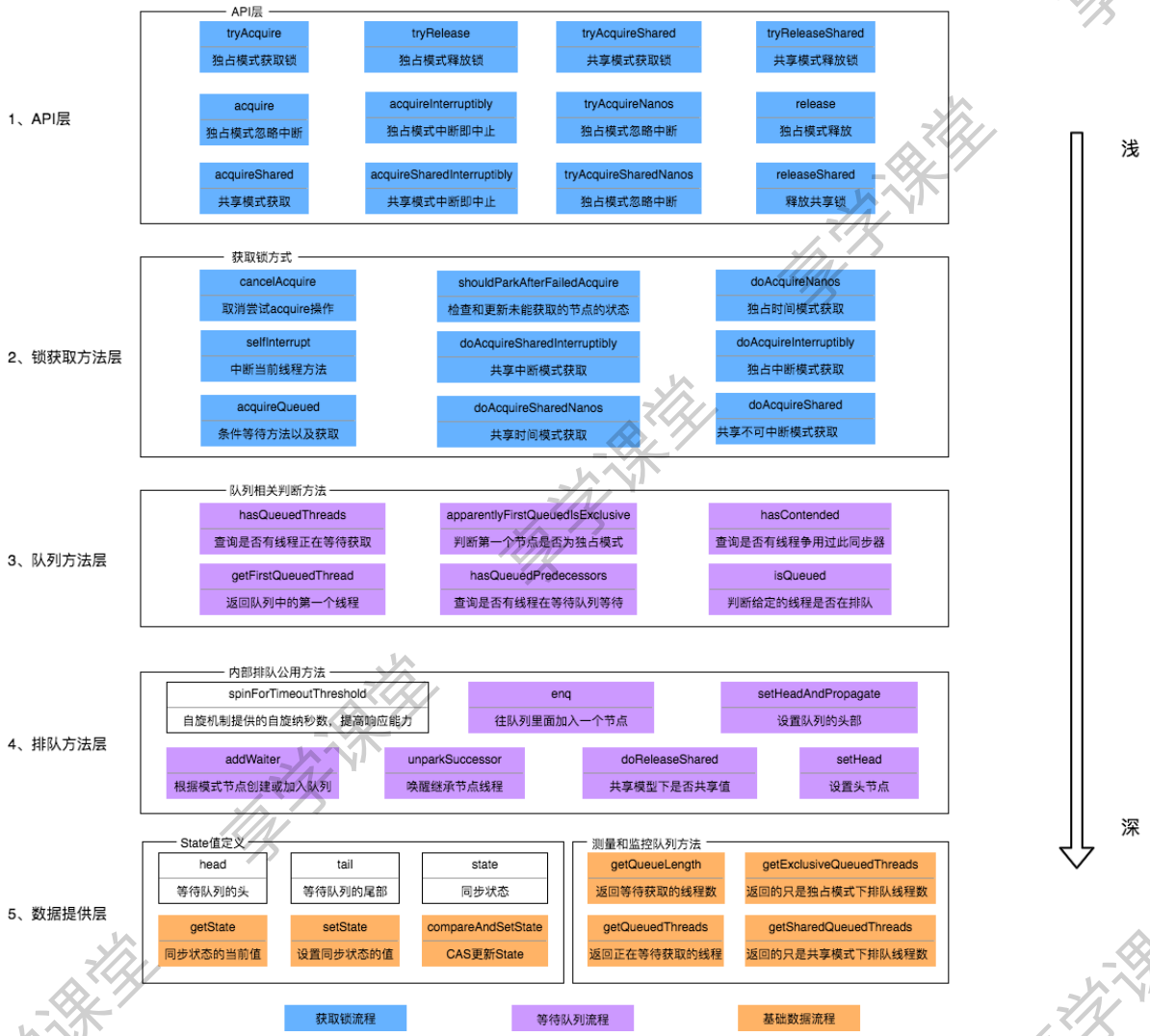
考生应该如何回答

什么是AQS

AQS即AbstractQueuedSynchronizer，是一个用于构建锁和同步器的框架。它能降低构建锁和同步器的工作量，还可以避免处理多个位置上发生的竞争问题。在基于AQS构建的同步器中，只可能在一个时刻发生阻塞，从而降低上下文切换的开销，并提高吞吐量。

AQS核心思想是，如果被请求的共享资源空闲，那么就将当前请求资源的线程设置为有效的工作线程，将共享资源设置为锁定状态；如果共享资源被占用，就需要一定的阻塞等待唤醒机制来保证锁分配。这个机制主要用的是CLH队列的变体实现的，将暂时获取不到锁的线程加入到队列中。

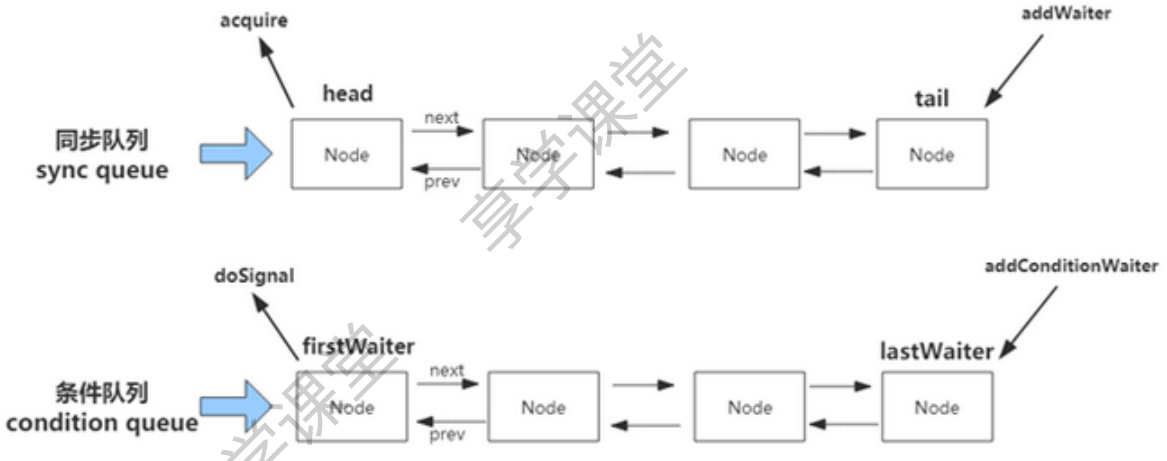
> CLH
>
> CLH指的是：三位创作者的名字简称:Craig、Landin and Hagersten (CLH)。是一种基于链表的可扩展、高性能、公平的自旋锁，申请线程仅仅在本地变量上自旋，它不断轮询前驱的状态，假设发现前驱释放了锁就结束自旋。
>
> AQS中的队列是CLH变体的虚拟双向队列（FIFO），AQS是通过将每条请求共享资源的线程封装成一个节点来实现锁的分配。



AQS支持独占锁（exclusive）和共享锁(share)两种模式。

1. 独占锁：只能被一个线程获取到(Reentrantlock)。
2. 共享锁：可以被多个线程同时获取(CountDownLatch,ReadWriteLock)。

> 无论是独占锁还是共享锁，本质上都是对AQS内部的一个变量state的获取。state是一个原子的int变量，用来表示锁状态、资源数等。



同步队列的作用是：当线程获取资源失败之后，就进入同步队列的尾部保持自旋等待，不断判断自己是否是链表的头节点，如果是头节点，就不断参试获取资源，获取成功后则退出同步队列。

条件队列是为Lock实现的一个基础同步器，并且一个线程可能会有多个条件队列，只有在使用了Condition才会存在条件队列。

AQS中包含一个内部类:Node。该内部类是一个双向链表，保存前后节点，然后每个节点存储了当前的状态waitStatus、当前线程thread。同步队列和条件队列都是由一个个Node组成的。

```
static final class Node {
        static final Node EXCLUSIVE = null;

        //当前节点由于超时或中断被取消
        static final int CANCELLED =  1;

        //表示当前节点的前节点被阻塞
        static final int SIGNAL    = -1;

        //当前节点在等待condition
        static final int CONDITION = -2;

        //状态需要向后传播
        static final int PROPAGATE = -3;

        volatile int waitStatus;

        volatile Node prev;
        volatile Node next;
        volatile Thread thread;

        Node nextWaiter;

        final boolean isShared() {
            return nextWaiter == SHARED;
        }

        final Node predecessor() throws NullPointerException {
            Node p = prev;
            if (p == null)
                throw new NullPointerException();
            else
                return p;
        }

        Node() {    // Used to establish initial head or SHARED marker
        }

        Node(Thread thread, Node mode) {     // Used by addWaiter
            this.nextWaiter = mode;
            this.thread = thread;
        }

        Node(Thread thread, int waitStatus) { // Used by Condition
            this.waitStatus = waitStatus;
            this.thread = thread;
        }
    }
```

独占模式下获取资源:

```
    public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

acquire(int arg)首先调用tryAcquire(arg)尝试直接获取资源，如果获取成功，因为与运算的短路性质，就不再执行后面的判断，直接返回。tryAcquire(int arg)的具体实现由子类负责。如果没有直接获取到资源，就将当前线程加入等待队列的尾部，并标记为独占模式，使线程在等待队列中自旋等待获取资源，直到获取资源成功才返回。如果线程在等待的过程中被中断过，就返回true，否则返回false。

如果acquireQueued(addWaiter(Node.EXCLUSIVE), arg)执行过程中被中断过，那么if语句的条件就全部成立，就会执行selfInterrupt();方法。因为在等待队列中自旋状态的线程是不会响应中断的，它会把中断记录下来，如果在自旋时发生过中断，就返回true。然后就会执行selfInterrupt()方法，而这个方法就是简单的中断当前线程Thread.currentThread().interrupt();其作用就是补上在自旋时没有响应的中断。

可以看出在整个方法中，最重要的就是acquireQueued(addWaiter(Node.EXCLUSIVE), arg)。我们首先看Node addWaiter(Node mode)方法，顾名思义，这个方法的作用就是添加一个等待者，根据之前对AQS中数据结构的分析，可以知道，添加等待者就是将该节点加入等待队列。

```
private Node addWaiter(Node mode) {
        Node node = new Node(Thread.currentThread(), mode);
        // Try the fast path of enq; backup to full enq on failure
        Node pred = tail;
        //尝试快速入队
        if (pred != null) { //队列已经初始化
            node.prev = pred;
            if (compareAndSetTail(pred, node)) {
                pred.next = node;
                return node; //快速入队成功后，就直接返回了
            }
        }
        //快速入队失败，也就是说队列都还没初始化，就使用enq
        enq(node);
        return node;
    }

    //执行入队
     private Node enq(final Node node) {
        for (;;) {
            Node t = tail;
            if (t == null) { // Must initialize
            //如果队列为空，用一个空节点充当队列头
                if (compareAndSetHead(new Node()))
                    tail = head;//尾部指针也指向队列头
            } else {
                //队列已经初始化，入队
                node.prev = t;
                if (compareAndSetTail(t, node)) {
                    t.next = node;
                    return t;//打断循环
                }
            }
        }
    }
```

```
final boolean acquireQueued(final Node node, int arg) {
        boolean failed = true;
        try {
            boolean interrupted = false;
            for (;;) {
                final Node p = node.predecessor();//拿到node的上一个节点
                //前置节点为head，说明可以尝试获取资源。排队成功后，尝试拿锁
                if (p == head && tryAcquire(arg)) {
                    setHead(node);//获取成功，更新head节点
                    p.next = null; // help GC
                    failed = false;
                    return interrupted;
                }
                //尝试拿锁失败后，根据条件进行park
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
    //获取资源失败后，检测并更新等待状态
    private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        int ws = pred.waitStatus;
        if (ws == Node.SIGNAL)
            /*
             * This node has already set status asking a release
             * to signal it, so it can safely park.
             */
            return true;
        if (ws > 0) {
            /*
             * Predecessor was cancelled. Skip over predecessors and
             * indicate retry.
             */
            do {
            //如果前节点取消了，那就往前找到一个等待状态的接待你，并排在它的后面
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            pred.next = node;
        } else {
            /*
             * waitStatus must be 0 or PROPAGATE.  Indicate that we
             * need a signal, but don't park yet.  Caller will need to
             * retry to make sure it cannot acquire before parking.
             */
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        return false;
    }
    //阻塞当前线程，返回中断状态
    private final boolean parkAndCheckInterrupt() {
        LockSupport.park(this);
        return Thread.interrupted();
    }
```

公平锁的实现

在并发环境中，每个线程在获取锁时会先查看此锁维护的等待队列，如果为空，或者当前线程是等待队列的第一个，就占有锁，否则就会加入到等待队列中，以后会按照FIFO的规则从队列中取到自己。公平锁的优点是等待锁的线程不会饿死。缺点是整体吞吐效率相对非公平锁要低，等待队列中除第一个线程以外的所有线程都会阻塞，CPU唤醒阻塞线程的开销比非公平锁大。

```
protected final boolean tryAcquire(int acquires) {
     final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {//状态为0表示可以加锁
                if (!hasQueuedPredecessors() && //hasQueuedPredecessors表示之前的线程是否有在排队的，这里加了！表示没有排队
                    compareAndSetState(0, acquires)) { //那么就去尝试cas state
                    setExclusiveOwnerThread(current); //如果cas成功设置排他线程为当前线程，表示成功得到锁
                    return true;
                }
            }
            else if (current == getExclusiveOwnerThread()) {//如果当前的排他线程是当前线程，表示是重入
                int nextc = c + acquires; //重入计数器增加
                if (nextc < 0)
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);//因为已经获得锁了，所以不用cas去设，直接设值就行
                return true;
            }
            return false;
}
```

非公平锁的实现

直接尝试占有锁，如果尝试失败，就再采用类似公平锁那种方式。非公平锁的优点是可以减少唤起线程的开销，整体的吞吐效率高，因为线程有几率不阻塞直接获得锁，CPU不必唤醒所有线程。缺点是处于等待队列中的线程可能会饿死，或者等很久才会获得锁。

```
final boolean nonfairTryAcquire(int acquires) {
    // 获取当前线程
    final Thread current = Thread.currentThread();
    // 获取当前state的值
    int c = getState();
    if (c == 0) {
      // 看看设置值是否能成功
            if (compareAndSetState(0, acquires)) {
           // 则将当前线程设置为独占线程
            setExclusiveOwnerThread(current);
            return true;
        }
    }
 // 返回由setExclusiveOwnerThread设置的最后一个线程；如果从不设置，则返回null
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0) // overflow
            throw new Error("Maximum lock count exceeded");
        // 设置state的值
        setState(nextc);
```

```
            return true;
        }
        return false;
    }
```

释放锁实现

释放锁代码分析：尝试释放此锁。如果当前线程是此锁的持有者，则保留计数将减少。 如果保持计数现在为零，则释放锁定。 如果当前线程不是此锁的持有者，则抛出IllegalMonitorStateException。

```
public void unlock() {
    sync.release(1);
}
```

sync.release(1) 调用的是AbstractQueuedSynchronizer中的release方法

```
## AbstractQueuedSynchronizer
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

分析tryRelease(arg)方法，tryRelease(arg)该方法调用的是ReentrantLock中

```
protected final boolean tryRelease(int releases) {
// 获取当前锁持有的线程数量和需要释放的值进行相减
    int c = getState() - releases;
    // 如果当前线程不是锁占有的线程抛出异常
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    // 如果此时c = 0就意味着state = 0，当前锁没有被任意线程占有
    // 将当前所的占有线程设置为空
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    // 设置state的值为 0
    setState(c);
    return free;
}
```

如果头节点不为空，并且waitStatus != 0，唤醒后续节点如果存在的话。这里的判断条件为什么是h != null && h.waitStatus != 0？因为h == null的话，Head还没初始化。初始情况下，head == null，第一个节点入队，Head会被初始化一个虚拟节点。所以说，这里如果还没来得及入队，就会出现head == null的情况。

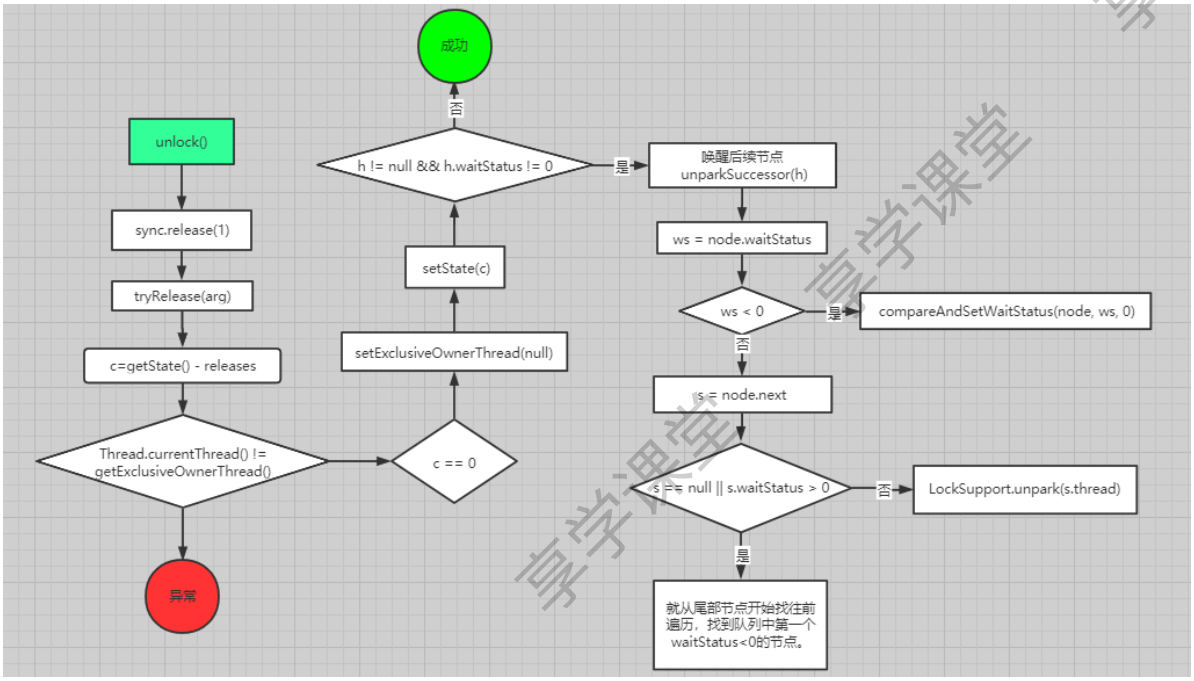
1. h != null && waitStatus == 0 表明后继节点对应的线程仍在运行中，不需要唤醒
2. h != null && waitStatus < 0 表明后继节点可能被阻塞了，需要唤醒

```
private void unparkSuccessor(Node node) {
// 获取头结点waitStatus
    int ws = node.waitStatus;
    if (ws < 0)
        compareAndSetWaitStatus(node, ws, 0);
// 获取当前节点的下一个节点
    Node s = node.next;
//如果下个节点是null或者下个节点被cancelled，就找到队列最开始的非cancelled的节点
    if (s == null || s.waitStatus > 0) {
        s = null;
        // 就从尾部节点开始找往前遍历，找到队列中第一个waitStatus<0的节点。
        for (Node t = tail; t != null && t != node; t = t.prev)
            if (t.waitStatus <= 0)
                s = t;
    }
  // 如果当前节点的下个节点不为空，而且状态<=0，就把当前节点唤醒
    if (s != null)
        LockSupport.unpark(s.thread);
}
```

为什么要从后往前找第一个非Cancelled的节点？

```
private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    enq(node);
    return node;
}
```

从此处可以看到，节点入队并不是原子操作，也就是说，node.prev = pred, compareAndSetTail(pred, node) 这两个地方可以看作Tail入队的原子操作，但是此时pred.next = node;还没执行，如果这个时候执行了unparkSuccessor方法，就没办法从前往后找了，所以需要从后往前找。还有一点原因，在产生CANCELLED状态节点的时候，先断开的是Next指针，Prev指针并未断开，因此也是必须要从后往前遍历才能够遍历完全部的Node。
所以，如果是从前往后找，由于极端情况下入队的非原子操作和CANCELLED节点产生过程中断开Next指针的操作，可能会导致无法遍历所有的节点。所以，唤醒对应的线程后，对应的线程就会继续往下执行。



4.6 ReentrantLock的实现原理

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《深入理解并发编程-AQS与JMM》

这道题想考察什么？

1. 是否了解并发相关的理论知识
2. 是否对于锁机制有个全面的理论认知
3. 是否对于AQS原理有自己的理解

考察的知识点

1. 锁的分类（公平锁、重入锁、重力度锁等等）
2. ReentrantLock实现方式与Synchronized实现方式的异同点

考生应该如何回答

Java中的大部分同步类（Lock、Semaphore、ReentrantLock等）都是基于队列同步器—AQS实现的。AQS原理见**《4.5 AQS原理》**。

在ReentrantLock中有一个抽象类Sync：

```
private final Sync sync;
abstract static class Sync extends AbstractQueuedSynchronizer {
    ...
}
```

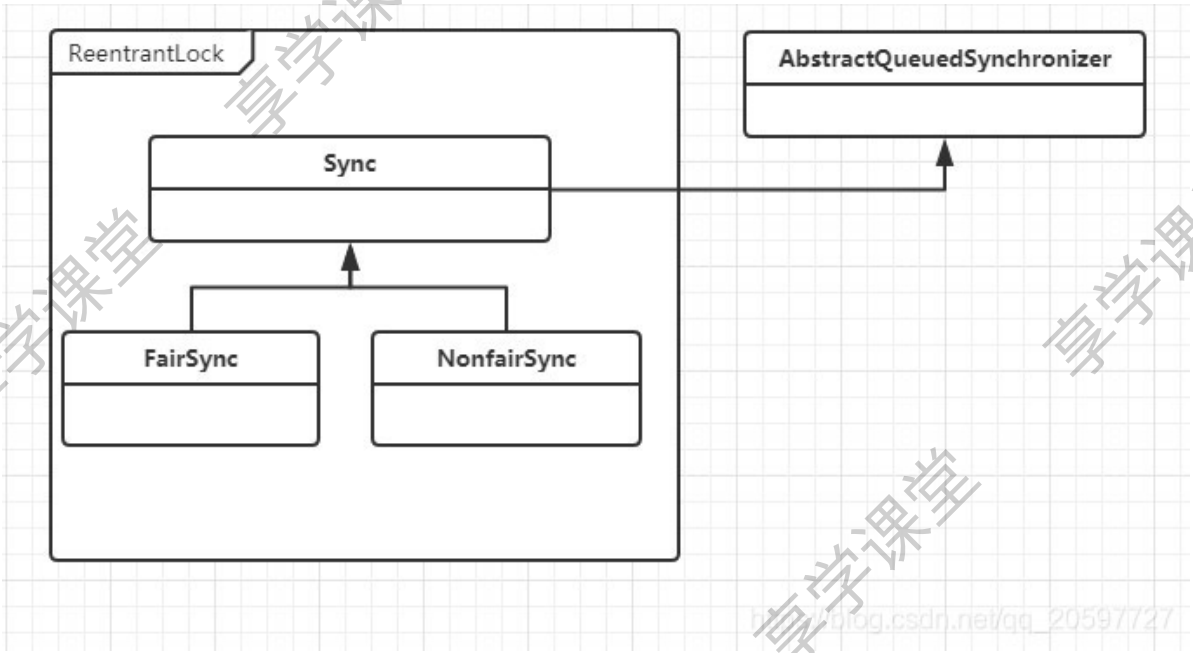
可以看到Sync就继承自AQS，而ReentrantLock的lock解锁、unlock释放锁等操作其实都是借助的sync来完成。

```
public void lock() {
    sync.lock();
}
public void unlock() {
    sync.release(1);
}
```

Sync是个抽象类，ReentrantLock根据传入构造方法的布尔型参数实例化出Sync的实现类FairSync和NonfairSync，分别表示公平锁和非公平锁。

```
public ReentrantLock() {
    sync = new NonfairSync();
}

public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
}
```

ReentrantLock与AQS的关系如下：



NonfairSync

在ReentrantLock的默认无参构造方法中，sync会被是实例化为：`NonfairSync` 表示非公平锁。

lock

```
static final class NonfairSync extends Sync {
    final void lock() {
        if (compareAndSetState(0, 1))
            setExclusiveOwnerThread(Thread.currentThread());
        else
            acquire(1);
    }
}
```

NonfairSync就是一个AQS。因此在执行lock时，会首先利用CAS（**《4.4 CAS无锁编程原理》**）尝试设置AQS的state为1。如果设置成功表示成功获取锁；否则表示其他线程已经占用，此时会使用`AQS#acquire`将尝试获取锁失败的线程放入AQS的等待队列进行等待并且将线程挂起。

unlock

lock获取锁需要对state进行加1，那么对于重入锁而言，重入一次就需要对state执行一次加1。这样子，在解锁的时候，每次unlock就对state减一，等到state的值为0的时候，才能唤醒下一个等待线程。

因此`ReentrantLock#unlock`，实际上就是执行了AQS的release(1)：

```
public void unlock() {
    sync.release(1);
}
```

FairSync

对于**NonfairSync** 而言，线程只要执行lock请求，就会马上尝试获取锁，不会管AQS当前管理的等待队列中是否存在正在等待的线程，这对于等待的线程不公平，因此NonfairSync表示非公平锁。

而**FairSync**表示公平锁，会在lock请求进行时，先判断AQS管理的等待队列中是否已经有正在等待的线程，有的话就不会尝试获取锁，直接进入等待队列，保证了公平性。此时`FairSync#lock`实际上执行的就是AQS的acquire

```
static final class FairSync extends Sync {
    final void lock() {
        acquire(1);
    }
}
```

4.7 Synchronized的原理以及与ReentrantLock的区别。（360）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《深入理解并发编程-AQS与JMM》

这道题想考察什么？

1. 是否了解并发相关的理论知识
2. 是否对于锁机制有个全面的理论认知
3. 是否对于AQS原理有自己的理解

考察的知识点

1. 锁的分类（公平锁、重入锁、重力度锁等等）
2. ReentrantLock实现方式与Synchronized实现方式的异同点

考生应该如何回答

Synchronized的原理见**《4.8 Synchronized在JDK1.6之后做了哪些优化》**。

ReentrantLock与Synchronized的区别，除了一个是Java类实现，一个是关键字之外，还包括：

	ReentrantLock	Synchronized
底层实现	通过 AQS 实现	通过 JVM 实现，其中 synchronized 又有多个类型的锁，除了重量级锁是通过 monitor 对象(操作系统mutex互斥原语)实现外，其它类型的通过对象头实现。
是否可重入	是	是
公平锁	是	否
非公平锁	是	是
锁的类型	悲观锁、显式锁	悲观锁、隐式锁(内置锁)
是否支持中断	是	否
是否支持超时等待	是	否
是否自动获取/释放锁	否	是

除此之外，ReenTrantLock相对于Synchronized还拥有自己的独有特性：

- ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。
- ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。
- ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。

4.8 volatile关键字干了什么？（什么叫指令重排）（字节跳动）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门2》与《深入理解并发编程-AQS与JMM》

这道题想考察什么？

是否了解volatile关键字与真实场景使用

考察的知识点

volatile关键字的概念在项目中使用与基本知识

考生应该如何回答

volatile是java提供的可以声明在成员属性前的一个关键字。在声明中包含此关键字的作用有：

保证内存可见性

可见性是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的。也就是一个线程修改的结果，另一个线程马上就能看到。

当对非volatile变量进行读写的时候，每个线程先从主内存拷贝变量到CPU缓存中，如果计算机有多个CPU，每个线程可能在不同的CPU上被处理，这意味着每个线程可以拷贝到不同的CPU cache中。

volatile变量不会被缓存在寄存器或者对其他处理器不可见的地方，保证了每次读写变量都从主内存中读，跳过CPU cache这一步。当一个线程修改了这个变量的值，新值对于其他线程是立即得知的。

禁止指令重排

指令重排序是JVM为了优化指令、提高程序运行效率，在不影响单线程程序执行结果的前提下，尽可能地提高并行度。指令重排序包括编译器重排序和运行时重排序。

latile变量禁止指令重排序。针对volatile修饰的变量，在读写操作指令前后会插入内存屏障，指令重排序时不能把后面的指令重排序到内存屏

示例说明：

```
示例说明：
double r = 2.1; //(1)
double pi = 3.14;//(2)
double area = pi*r*r;//(3)
```

虽然代码语句的定义顺序为1->2->3，但是计算顺序1->2->3与2->1->3对结果并无影响，所以编译时和运行时可以根据需要对1、2语句进行重排序。

指令重排带来的问题

如果一个操作不是原子的，就会给JVM留下重排的机会。

```
线程A中
{
    context = loadContext();
    inited = true;
}

线程B中
{
    if (inited)
        fun(context);
}
```

如果线程A中的指令发生了重排序，那么B中很可能就会拿到一个尚未初始化或尚未初始化完成的context,从而引发程序错误。

禁止指令重排的原理

volatile关键字提供内存屏障的方式来防止指令被重排，编译器在生成字节码文件时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。

JVM内存屏障插入策略：

在每个volatile写操作的前面插入一个StoreStore屏障；

在每个volatile写操作的后面插入一个StoreLoad屏障；

在每个volatile读操作的后面插入一个LoadLoad屏障；

在每个volatile读操作的后面插入一个LoadStore屏障。

指令重排在双重锁定单例模式中的影响
基于双重检验的单例模式(懒汉型)

```
public class Singleton3 {
    private static Singleton3 instance = null;

    private Singleton3() {}

    public static Singleton3 getInstance() {
        if (instance == null) {
            synchronized(Singleton3.class) {
                if (instance == null)
                    instance = new Singleton3();// 非原子操作
            }
        }

        return instance;
    }
}
```

instance= new Singleton()并不是一个原子操作，其实际上可以抽象为下面几条JVM指令：

```
memory =allocate();    //1：分配对象的内存空间
ctorInstance(memory);  //2：初始化对象
instance =memory;      //3：设置instance指向刚分配的内存地址
```

上面操作2依赖于操作1，但是操作3并不依赖于操作2。所以JVM是可以针对它们进行指令的优化重排序的，经过重排序后如下：

```
memory =allocate();    //1：分配对象的内存空间
instance =memory;      //3：instance指向刚分配的内存地址，此时对象还未初始化
ctorInstance(memory);  //2：初始化对象
```

指令重排之后，instance指向分配好的内存放在了前面，而这段内存的初始化被排在了后面。在线程A执行这段赋值语句，在初始化分配对象之前就已经将其赋值给instance引用，恰好另一个线程进入方法判断instance引用不为null，然后就将其返回使用，导致出错。

volatile解决重排

用volatile关键字修饰instance变量，使得instance在读、写操作前后都会插入内存屏障，避免重排序。

```
public class Singleton3 {
    private static volatile Singleton3 instance = null;

    private Singleton3() {}

    public static Singleton3 getInstance() {
        if (instance == null) {
            synchronized(Singleton3.class) {
                if (instance == null)
                    instance = new Singleton3();
            }
        }
        return instance;
    }
}
```

4.9 volatile 能否保证线程安全？在DCL上的作用是什么？

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门2》与《深入理解并发编程-AQS与JMM》

这道题想考察什么？

1. 是否了解Java并发编程的相关知识？
2. 对象创建的过程

考察的知识点

1. volatile的原理
2. 编译优化

考生应该如何回答

volatile无法保证线程安全，只能保证变量的可见性，并不能保证变量操作的原子性。

原子性指的是一个或者多个操作在 CPU 执行的过程中不被中断的特性。

```
public class VolatileTest {

    public volatile static int count = 0;

    public static void main(String [] args){
        //开启5个线程
      for(int i = 0;i < 5; i++){
         new Thread(new Runnable() {

             @Override
```

```
            public void run() {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                //让count的值自增100次
                for(int j = 0;j < 100;j++){
                    count++;
                }
            }
        }).start();
    }

      try {
          Thread.sleep(3000);
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
      System.out.println("count= " + count);
  }
}
```

上面代码我们开启5个线程，每个线程当中让静态变量count自增100次。执行之后会发现，最终count的结果值未必是500，有可能小于500。

出现上面情况的原因是因为volatile没有保证原子性。例如A线程获取到count的值为2，此时主存与工作内存数据一致，然后我们执行自增操作，count的值为3，但是主存中的值很有可能被其他线程更新为了8或者其他数目，如果A线程执行更新主存，那数目相当于往下降了。

volatile在单例中的作用

```
public class SingletonClass {

  private volatile static SingletonClass instance = null;

  public static SingletonClass getInstance() {
    if (instance == null) {
      synchronized (SingletonClass.class) {
        if(instance == null) {
          instance = new SingletonClass();
        }
      }
    }
    return instance;
  }
  private SingletonClass() {
  }
}
```

上述代码中，我们使用到了双重检测和volatile，为什么在双重检测的基础上还需要加上volatile这是因为我们对象创建的部分，可能会因为指令重排发生变化。一般情况下包括三步：

1. 分配内存
2. 初始化对象
3. 将内存地址赋值给引用

如果发生了指令重排可能会导致第二步内容和第三步内容顺序发生变化，即还没初始化的对象已经赋值给引用，再执行相关操作会发生各类异常问题。

总结

因为volatile不能保证变量操作的原子性，所以试图通过volatile来保证线程安全性是不靠谱的。

volatile在DCL上的作用是防止对象发生指令重排而引起的异常问题。

4.10 volatile和synchronize有什么区别？（B站 小米 京东）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门2》

这道题想考察什么？

是否了解Java并发编程的相关知识？

考察的知识点

1. volatile的原理
2. synchronize的原理

考生应该如何回答

- volatile 只能作用于变量，synchronized 可以作用于变量、方法、对象。
- volatile 只保证了可见性和有序性，无法保证原子性，synchronized 可以保证线程间的有序性（个人猜测是无法保证线程内的有序性，即线程内的代码可能被 CPU 指令重排序）、原子性和可见性。
- volatile 线程不阻塞，synchronized 线程阻塞。
- volatile 本质是告诉 jvm 当前变量在寄存器中的值是不安全的需要从内存中读取；sychronized 则是锁定当前变量，只有当前线程可以访问到该变量其他线程被阻塞。
- volatile标记的变量不会被编译器优化， synchronized标记的变量可以被编译器优化。

4.11 死锁的场景和解决方案 腾讯

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《并发基础与CAS基本原理》

这道题想考察什么？

是否真正了解死锁的定义？是否掌握死锁的排查与解决

考察的知识点

并发编程 死锁

考生应该如何回答

死锁的定义

死锁是指两个或两个以上的进程在执行过程中，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。

危害

1、线程不工作了，但是整个程序还是活着的

2、没有任何的异常信息可以供我们检查。

3、一旦程序发生了发生了死锁，是没有任何的办法恢复的，只能重启程序，对正式已发布程序来说，这是个很严重的问题。

死锁的发生必须具备以下四个必要条件。

1. 互斥条件：指进程对所分配到的资源进行排它性使用，即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，则请求者只能等待，直至占有资源的进程用毕释放。
2. 请求和保持条件：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. 不剥夺条件：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。
4. 环路等待条件：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，…，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

理解了死锁的原因，尤其是产生死锁的四个必要条件，就可以最大可能地避免、预防和解除死锁。只要打破四个必要条件之一就能有效预防死锁的发生。

1. 打破互斥条件：改造独占性资源为虚拟资源，大部分资源已无法改造。
2. 打破不可抢占条件：当一进程占有一独占性资源后又申请一独占性资源而无法满足，则退出原占有的资源。
3. 打破占有且申请条件：采用资源预先分配策略，即进程运行前申请全部资源，满足则运行，不然就等待，这样就不会占有且申请。
4. 打破循环等待条件：实现资源有序分配策略，对所有设备实现分类编号，所有进程只能采用按序号递增的形式申请资源。

避免死锁常见的算法有：**有序资源分配法**、**银行家算法**等。

有序资源分配法

有序资源分配法是预防死锁的一种算法，按某种规则对系统中的所有资源统一编号，申请时必须以上升的次序。

例如做饭时候盐为1，酱油为2等等。如果A、B两个厨师同时做饭，使用资源顺序分别为：

A：申请顺序1->2

B：申请顺序2->1

此时，A在拿着盐的同时要使用酱油，但是由于酱油被B持有，两人谁也不让谁。此时形成环路条件，造成死锁 。但是采用有序资源分配法，则：

A：申请顺序1->2

B：申请顺序1->2

A如果先获取到盐，那么B此时只能等待。这样就破坏了环路条件，避免了死锁的发生。

总结

死锁是必然发生在多操作者（M>=2个）情况下，争夺多个资源（N>=2个，且N<=M）才会发生这种情况。很明显，单线程自然不会有死锁，只有B一个去，不要2个，打十个都没问题；单资源呢？只有13，A和B也只会产生激烈竞争，打得不可开交，谁抢到就是谁的，但不会产生死锁。

4.12 锁分哪几类？

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门1》

这道题想考察什么？

是否了解并发相关锁的知识？

考察的知识点

1. 锁的分类和概念
2. 如何运用锁解决并发问题

考生应该如何回答

Java锁的种类

- 乐观锁/悲观锁
- 独享锁/共享锁
- 互斥锁/读写锁
- 可重入锁
- 公平锁/非公平锁
- 分段锁
- 偏向锁/轻量级锁/重量级锁
- 自旋锁

以上是一些锁的名词，这些分类并不是全是指锁的状态，有的指锁的特性，有的指锁的设计。

乐观锁/悲观锁

乐观锁与悲观锁并不是特指某两种类型的锁，是人们定义出来的概念或思想，主要是指看待并发同步的角度。

- 乐观锁：获取数据时认为不会被其他线程修改，所以不会上锁，但是在更新的时候会判断其他线程是否修改此数据，如果被其他线程修改，则会发生自旋。
- 悲观锁：总是假设最坏的情况，获取数据时都认为其他线程会修改，因此在获取数据时都会上锁，这样保证其他线程需要等待获取锁的线程处理完成并且释放锁。

乐观锁适用于频繁读取的场景，因为不会上锁，因此可以提高吞吐量。在Java中java.util.concurrent.atomic包下面的原子变量类就是基于乐观锁的一种实现方式CAS(Compare and Swap 比较并交换)实现的。

悲观锁适合写操作较多的场景，synchronized关键字的实现就是悲观锁。

独享锁/共享锁

- 独享锁是指该锁一次只能被一个线程所持有。
- 共享锁是指该锁可被多个线程所持有。

ReentrantLock是独享锁。但是对于Lock的另一个实现读写锁ReadWriteLock，读锁是共享锁，而写锁则是独享锁。

互斥锁/读写锁

上面讲的独享锁/共享锁就是一种广义的说法，互斥锁/读写锁就是具体的实现。

- 互斥锁在Java中的具体实现就是ReentrantLock。
- 读写锁在Java中的具体实现就是ReadWriteLock。

可重入锁

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，在进入内层方法会自动获取锁。synchronized与ReetrantLock都是可重入锁。可重入锁的一个好处就是可以在一定程度避免死锁：

```
synchronized void setA() throws Exception{
    Thread.sleep(1000);
    setB();
}

synchronized void setB() throws Exception{
    Thread.sleep(1000);
}
```

上述代码中，如果synchronized不是可重入锁的话，`setA`首先获取锁，在此方法还未释放锁的情况下，调用`setB`也需要获取相同的对象锁，此时会造成死锁。

公平锁/非公平锁

公平锁是指多个线程按照申请锁的顺序获取锁，非公平锁则是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。非公平锁的优点在于吞吐量比公平锁大，但是也有可能会造成优先级反转或者饥饿现象。

Java中ReetrantLock可以通过构造函数指定该锁是否是公平锁，默认是非公平锁。而synchronized则是非公平锁。

分段锁

分段锁其实是一种锁的设计，并不是具体的一种锁。比如`ConcurrentHashMap`，其并发的实现就是通过分段锁的形式来实现高效的并发操作。

ConcurrentHashMap中的分段锁封装为Segment，它本身也是类似于HashMap的结构，其内部拥有一个Entry数组，数组中的每个元素又是一个链表；同时又是一个ReentrantLock（Segment继承了ReentrantLock）。

当需要put元素的时候，并不是对整个HashMap进行加锁，而是先通过hashcode来知道他要放在哪一个分段中，然后对这个分段进行加锁，所以当多线程put的时候，只要不是放在一个分段中，就实现了真正的并行的插入。

分段锁的设计目的是细化锁的粒度，当操作不需要更新整个数组的时候，就仅仅针对数组中的一项进行加锁操作。

偏向锁/轻量级锁/重量级锁

这三种锁是指锁的状态，偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。降低获取锁的代价。

轻量级锁是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。

重量级锁是指当锁为轻量级锁的时候，另一个线程虽然是自旋，但自旋不会一直持续下去，当自旋一定次数的时候，还没有获取到锁，就会进入阻塞，该锁膨胀为重量级锁。重量级锁会让他申请的线程进入阻塞，性能降低。

自旋锁

在Java中，自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗CPU。

4.13 ThreadLocal是什么？

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门2》

这道题想考察什么？

是否了解ThreadLocal与真实场景使用，是否熟悉ThreadLocal

考察的知识点

ThreadLocal的概念在项目中使用与基本知识

考生应该如何回答

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<String> threadLocal = new ThreadLocal<>();
threadLocal.set("享学");
System.out.println("主线程获取变量："+threadLocal.get());
Thread thread = new Thread() {
    @Override
    public void run() {
        super.run();
        System.out.println("子线程获取变量："+ threadLocal.get());
        threadLocal.set("教育");
        System.out.println("子线程获取变量："+ threadLocal.get());
    }
};
```

在上述代码中，主线程输出：享学，子线程第一次输出：null，第二次输出教育。ThreadLocal相当于提供了一种线程隔离，将变量与线程相绑定。

set

通过`ThreadLocal#set`设置线程本地变量，set的实现为：

```
public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}
```

通过Thread.currentThread()方法获取了当前的线程引用，并传给了getMap(Thread)方法获取一个ThreadLocalMap的实例。

```
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}
```

可以看到getMap(Thread)方法直接返回Thread实例的成员变量threadLocals。它的定义在Thread内部：

```
public class Thread implements Runnable {
    ThreadLocal.ThreadLocalMap threadLocals = null;
}
```

每个Thread里面都有一个ThreadLocal.ThreadLocalMap成员变量，也就是说每个线程通过ThreadLocal.ThreadLocalMap与ThreadLocal相绑定，这样可以确保每个线程访问到变量的都是本线程自己的。

获取了ThreadLocalMap实例以后，如果它不为空则调用ThreadLocalMap.ThreadLocalMap 的set方法设值；若为空则调用ThreadLocal 的createMap方法new一个ThreadLocalMap实例并赋给Thread.threadLocals。

```
void createMap(Thread t, T firstValue) {
    // this = ThreadLocal
    t.threadLocals = new ThreadLocalMap(this, firstValue);
}
```

get

而ThreadLocal 的 get 方法，源码如下：

```
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null)
            return (T)e.value;
    }
    return setInitialValue();
}
```

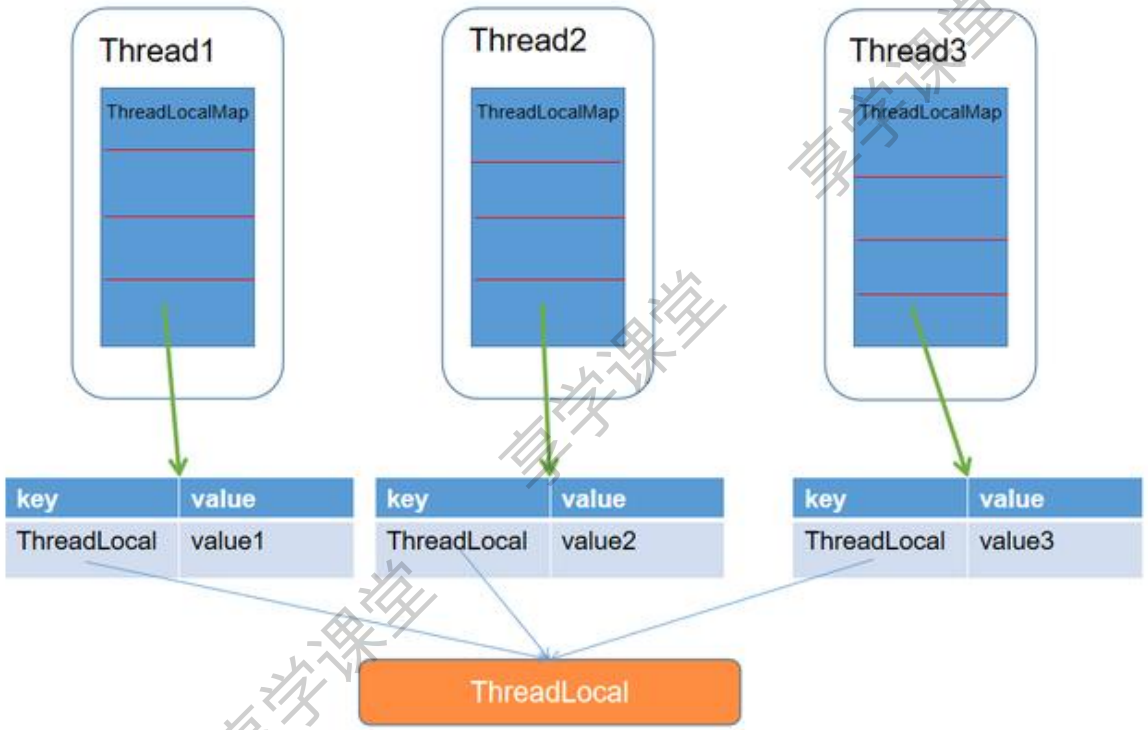
同样通过Thread.currentThread()方法获取了当前的线程引用，并传给了getMap(Thread)方法获取一个ThreadLocalMap的实例。而如果从ThreadLocalMap未能找到当前线程的变量则返回`setInitialValue`。

```
private T setInitialValue() {
    T value = initialValue();
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
    return value;
}
```

在setInitialValue中首先调用 initialValue()方法来获得一个value，然后执行`ThreadLocal#set`同样的处理并返回这个value，也就是说可以通过重写ThreadLocal的initialValue方法能够实现在set变量值之前，使用get获取的就是这个initialValue返回的结果。

```
ThreadLocal<String> threadLocal = new ThreadLocal<String>(){
    @Nullable
    @Override
    protected String initialValue() {
        return "享学";
    }
};
// 享学
String value = threadLocal.get();
```

在set/get中其实就是借助ThreadLocalMap实现线程与本地变量的绑定与获取。每个线程都有自己的一个ThreadLocalMap，ThreadLocalMap是一个映射集合，以ThreadLocal为key。



ThreadLocal简化的伪代码为：

```
class Thread extends Thread {
    ThreadLocalMap threadLocals;
}

class ThreadLocal<T> {
    public void set(T t) {
        Thread thread = Thread.currentThread();
        thread.threadLocals.put(this, t);
    }

    public T get() {
        Thread thread = Thread.currentThread();
        thread.threadLocals.get(this);
    }
}
```

4.14 Java多线程对同一个对象进行操作（字节跳动）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门1》

这道题想考察什么？

是否了解Java多线程对同一个对象进行操作与真实场景使用，是否熟悉Java多线程对同一个对象进行操作？

考察的知识点

Java多线程对同一个对象进行操作的概念在项目中使用与基本知识

考生应该如何回答

在多线程环境下，多个线程操作同一对象，本质上就是线程安全问题。因此为了应对线程安全需要对多线程操作的对象加锁。

例如当我们遇到需求：实现三个窗口同时出售20张票。

程序分析：

1、票数要使用一个静态的值。

2、为保证不会出现卖出同一张票，要使用同步锁。

3、设计思路：创建一个站台类Station，继承Thread，重写run方法，在run方法内部执行售票操作。

售票要使用同步锁：即有一个站台卖这张票时，其他站台要等待这张票卖完才能继续卖票！

```
package com.multi_thread;

//站台类
public class Station extends Thread {
    // 通过构造方法给线程名字赋值
    public Station(String name) {
        super(name);// 给线程起名字
    }

    // 为了保持票数的一直，票数要静态
    static int tick = 20;
    // 创建一个静态钥匙
    static Object ob = "aa";// 值是任意的

    @Override
    public void run() {
        while (tick > 0) {
            // 这个很重要，必须使用一个锁，进去的人会把钥匙拿在手上，出来后把钥匙让出来
            synchronized (ob) {
                if (tick > 0) {
                    System.out.println(getName() + "卖出了第" + tick + "张票");
                    tick--;
                } else {
                    System.out.println("票卖完了");
                }
            }
            try {
                // 休息一秒钟
                sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
        }
    }
}
```

```
package com.multi_thread;

public class MainClass {
    // java多线程同步所的使用
    // 三个售票窗口同时出售10张票
    public static void main(String[] args) {
        // 实例化站台对象，并为每一个站台取名字
        Station station1 = new Station("窗口1");
        Station station2 = new Station("窗口2");
        Station station3 = new Station("窗口3");
        // 让每一个站台对象各自开始工作
        station1.start();
        station2.start();
        station3.start();
    }
}
```

```
程序运行结果：

窗口1卖出了第20张票
窗口3卖出了第19张票
窗口2卖出了第18张票
窗口2卖出了第17张票
窗口3卖出了第16张票
窗口1卖出了第15张票
窗口1卖出了第14张票
窗口3卖出了第13张票
窗口2卖出了第12张票
窗口1卖出了第11张票
窗口3卖出了第10张票
窗口2卖出了第9张票
窗口1卖出了第8张票
窗口3卖出了第7张票
窗口2卖出了第6张票
窗口1卖出了第5张票
窗口3卖出了第4张票
窗口2卖出了第3张票
窗口3卖出了第2张票
窗口1卖出了第1张票
```

4.15 线程生命周期，线程可以多次调用start吗？会出现什么问题？为什么不能多次调用start？

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门1》

这道题想考察什么？

是否了解Java并发线程的相关知识

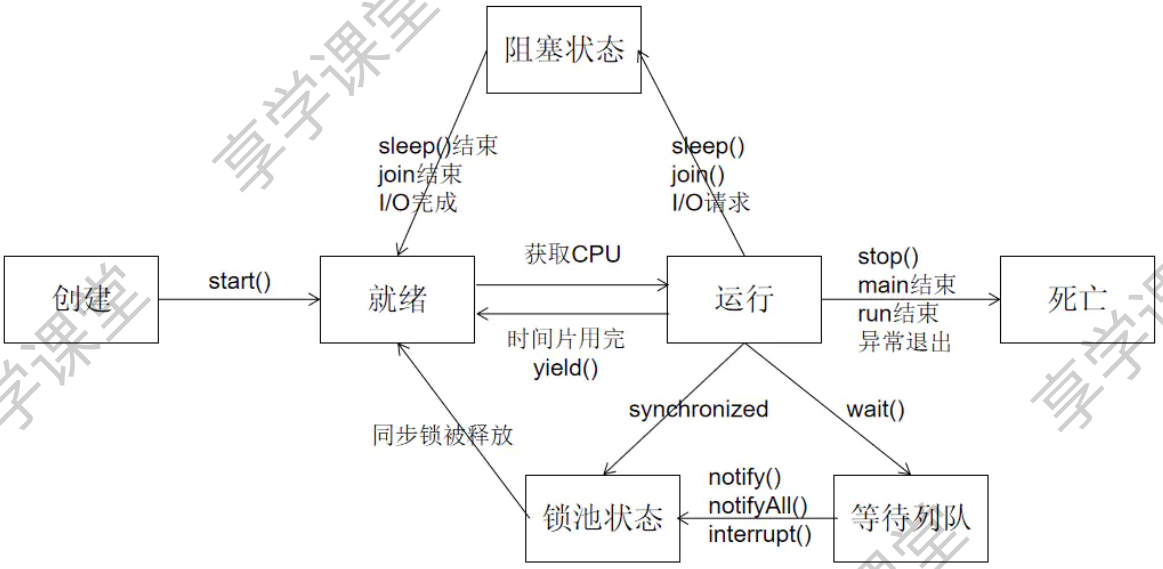
考察的知识点

线程生命周期及变化

考生应该如何回答

线程生命周期中重要的状态

- 新建 New；
- 就绪 Runnable
- 运行 Running
- 阻塞 Blocked
- 死亡 Dead



线程生命周期简图

新建 new

```
public class CThread extends Thread{
    @Override
        public void run() {

        }
}
//新建就是new出对象
CThread thread = new CThread();
```

当程序使用new关键字创建了一个线程之后，该线程就处于一个新建状态（初始状态），此时它和其他Java对象一样，仅仅由Java虚拟机为其分配了内存，并初始化了其成员变量值。此时的线程对象没有表现出任何线程的动态特征，程序也不会执行线程的线程执行体。

就绪 Runnable

当线程对象调用了Thread.start()方法之后，该线程处于就绪状态，Java虚拟机会为其创建方法调用栈和程序计数器，处于这个状态的线程并没有开始运行，它只是表示该线程可以运行了。从start()源码中看出，start后添加到了线程列表中，接着在native层添加到VM中，至于该线程何时开始运行，取决于JVM里线程调度器的调度(如果OS调度选中了，就会进入到运行状态)。回看一下下面start方法源码：

```
public synchronized void start() {
        /**
         * This method is not invoked for the main method thread or "system"
         * group threads created/set up by the VM. Any new functionality added
         * to this method in the future may have to also be added to the VM.
         *
         * A zero status value corresponds to state "NEW".
         */
        // Android-changed: throw if 'started' is true
        if (threadStatus != 0 || started)
            throw new IllegalThreadStateException();

        /* Notify the group that this thread is about to be started
         * so that it can be added to the group's list of threads
         * and the group's unstarted count can be decremented. */
         //通知组此线程即将启动，以便将其添加到组的线程列表中，并且可以减少组的未启动计数。
        group.add(this);
        started = false;
        try {
            nativeCreate(this, stackSize, daemon);
            started = true;
        } finally {
            try {
                if (!started) {
                    group.threadStartFailed(this);
                }
            } catch (Throwable ignore) {
                /* do nothing. If start0 threw a Throwable then
                  it will be passed up the call stack */
            }
        }
    }
```

C/C++中的nativeCreate的源码

```
static void Thread_nativeCreate(JNIEnv* env, jclass, jobject java_thread,
                                jlong stack_size, jboolean daemon) {
  Thread::CreateNativeThread(env, java_thread, stack_size, daemon == JNI_TRUE);
}
```

C/C++中的CreateNativeThread

```
void Thread::CreateNativeThread(JNIEnv* env, jobject java_peer, size_t
stack_size, bool is_daemon) {
  Thread* self = static_cast<JNIEnvExt*>(env)->self;
  Runtime* runtime = Runtime::Current();

  ...
  Thread* child_thread = new Thread(is_daemon);
```

```
  child_thread->tlsPtr_.jpeer = env->NewGlobalRef(java_peer);
  stack_size = FixStackSize(stack_size);

  env->SetLongField(java_peer, WellKnownClasses::java_lang_Thread_nativePeer,
                    reinterpret_cast<jlong>(child_thread));

  std::unique_ptr<JNIEnvExt> child_jni_env_ext(
      JNIEnvExt::Create(child_thread, Runtime::Current()->GetJavaVM()));

  int pthread_create_result = 0;
  if (child_jni_env_ext.get() != nullptr) {
    pthread_t new_pthread;
    pthread_attr_t attr;
    child_thread->tlsPtr_.tmp_jni_env = child_jni_env_ext.get();
    //创建线程
    pthread_create_result = pthread_create(&new_pthread,
                          &attr, Thread::CreateCallback, child_thread);

    if (pthread_create_result == 0) {
      child_jni_env_ext.release();
      return;
    }
  }

  ...
}
```

C/C++中的pthread_create，pthread_create的分析暂时不分析，涉及到Linux知识代深入了解再分析，先说说pthread_create的参数

- 原型：int pthread_create（（pthread_t thread, pthread_attr_t *attr, void *（start_routine）（void *）, void *arg）
- 头文件：#include
- 输入参数：thread：线程标识符； attr：线程属性设置； start_routine：线程函数的起始地址； -arg：传递给start_routine的参数；
- 返回值：成功则返回0；出错则返回-1。
- 功能：创建线程，并调用线程起始地址所指向的函数start_routine。

运行 Running

如果处于就绪状态的线程获得了CPU资源，就开始执行run方法的线程执行体，则该线程处于运行状态。run方法的那里呢？其实run也是在native线程中。源码如下：

```
status_t Thread::run(const char* name, int32_t priority, size_t stack)
{
    Mutex::Autolock _l(mLock);
    //保证只会启动一次
    if (mRunning) {
        return INVALID_OPERATION;
    }
    ...
    mRunning = true;

    bool res;
```

```
    if (mCanCallJava) {
        //还能调用Java代码的Native线程
        res = createThreadEtc(_threadLoop,
                this, name, priority, stack, &mThread);
    } else {
        //只能调用C/C++代码的Native线程
        res = androidCreateRawThreadEtc(_threadLoop,
                this, name, priority, stack, &mThread);
    }

    if (res == false) {
        ...//清理
        return UNKNOWN_ERROR;
    }
    return NO_ERROR;
}
```

mCanCallJava在Thread对象创建时，在构造函数中默认设置mCanCallJava=true.

- 当mCanCallJava=true,则代表创建的是不仅能调用C/C++代码，还能能调用Java代码的Native线程
- 当mCanCallJava=false,则代表创建的是只能调用C/C++代码的Native线程。

关于createThreadEtc和androidCreateRawThreadEtc方法都不一一列出来了，感兴趣的自己查检源码了解。

从start方法进入nativeCreate经过层层调用，最终都会进入clone系统调用，这是linux创建线程或进程的通用接口。Native线程中是否可以执行Java代码的区别，在于通过javaThreadShell()方法从而实现在_threadLoop()执行前后增加分别将当前线程增加hook到虚拟机和从虚拟机移除的功能。调用过程，顺序为：

1.Thread.run
2.createThreadEtc
3.androidCreateThreadEtc
4.javaCreateThreadEtc
5.androidCreateRawThreadEtc
6.javaThreadShell
7.javaAttachThread
8._threadLoop
9.javaDetachThread

阻塞 Blocked

阻塞状态是线程因为某种原因放弃CPU使用权，暂时停止运行。直到线程进入就绪状态，才有机会转到运行状态。阻塞的情况大概三种：

1、**等待阻塞：**运行的线程执行wait()方法，JVM会把该线程放入等待池中。(wait会释放持有的锁)
2、**同步阻塞：**运行的线程在获取对象的同步锁时，若该同步锁被别的线程占用，则JVM会把该线程放入锁池中。
3、**其他阻塞：**运行的线程执行sleep()或join()方法，或者发出了I/O请求时，JVM会把该线程置为阻塞状态。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。（注意,sleep是不会释放持有的锁）。

线程睡眠：Thread.sleep(long millis)方法，使线程转到阻塞状态。millis参数设定睡眠的时间，以毫秒为单位。当睡眠结束后，就转为就绪（Runnable）状态。sleep()平台移植性好。
线程等待：Object类中的wait()方法，导致当前的线程等待，直到其他线程调用此对象的 notify() 方法或 notifyAll() 唤醒方法。这个两个唤醒方法也是Object类中的方法，行为等价于调用 wait(0) 一样。唤醒线程后，就转为就绪（Runnable）状态。
线程让步：Thread.yield() 方法，暂停当前正在执行的线程对象，把执行机会让给相同或者更高优先级的线程。
线程加入：join()方法，等待其他线程终止。在当前线程中调用另一个线程的join()方法，则当前线程转入阻塞状态，直到另一个进程运行结束，当前线程再由阻塞转为就绪状态。
线程I/O：线程执行某些IO操作，因为等待相关的资源而进入了阻塞状态。比如说监听system.in，但是尚且没有收到键盘的输入，则进入阻塞状态。
线程唤醒：Object类中的notify()方法，唤醒在此对象监视器上等待的单个线程。如果所有线程都在此对象上等待，则会选择唤醒其中一个线程，选择是任意性的，并在对实现做出决定时发生。类似的方法还有一个notifyAll()，唤醒在此对象监视器上等待的所有线程。

死亡 Dead

线程会以以下三种方式之一结束，结束后就处于死亡状态:

- run()方法执行完成，线程正常结束。
- 线程抛出一个未捕获的Exception或Error。
- 直接调用该线程的stop()方法来结束该线程——该方法容易导致死锁，通常不推荐使用。

线程多次启动

Java线程是不允许启动多次的，第二次调用必然会抛出IllegalThreadStateException。 根据线程生命周期可知，线程初始状态为NEW，此状态不能由其他状态转变而来。

4.16 什么是守护线程？你是如何退出一个线程的？

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门1》

这道题想考察什么？

是否了解守护线程与真实场景使用，是否熟悉线程退出该如何操作的本质区别？

考察的知识点

守护线程与线程退出的概念在项目中使用与基本知识

考生应该如何回答

守护线程

在开发过程中，直接创建的普通线程为用户线程，而另一种线程，也就是守护线程，通过`setDaemon(true)`将一个普通用户线程设置为守护线程。

守护线程，也叫Daemon线程，它是一种支持型、服务型线程，主要被用作程序中后台调度以及支持性工作，跟上层业务逻辑基本不挂钩。Java中垃圾回收线程就是一个典型的Daemon线程。

```
public class DaemonThreadTest {
    public static void main(String[] args) {
        // 创建线程
        Thread daemonThread = new Thread("Daemon") {
            @Override
            public void run() {
                super.run();
                // 循环执行5次(休眠一秒，输出一句话)
                for (int i = 0; i < 5; i++) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    System.out.println("守护线程->" +
Thread.currentThread().getName() + "正在执行");                          }

            }
        };
        // 设置为守护线程，必须在线程启动start()方法之前设置，否则会抛出
IllegalThreadStateException异常
        daemonThread.setDaemon(true);
        // 启动守护线程
        daemonThread.start();
        // 这里是主线程逻辑
        // 循环执行3次(休眠一秒，输出一句话)
        for (int i = 0; i < 3; i++) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("普通线程->" + Thread.currentThread().getName() +
"正在执行");
        }
    }
}
```

在主线程里创建守护线程，然后一起执行，主线程的业务逻辑是每隔1s，输出一句log，总共循环三次，而守护线程里也是隔1s输出一句log，不过想要循环5次。其中需要注意的是设置为守护线程，必须在线程启动start()方法之前设置，否则会抛出IllegalThreadStateException异常，意思就是运行中的线程不能设置成守护线程的。最后我们运行一下main方法，看看控制台的输出结果如下：

```
控制台的输出普通线程->
main正在执行守护线程->
Daemon正在执行守护线程->
Daemon正在执行普通线程->
main正在执行守护线程->
Daemon正在执行普通线程->
main正在执行BUILD SUCCESSFUL in 3s2 actionable tasks: 2 executed
```

从上面的日志来看，两个线程都只是输出了3次，然后就执行结束，这其实就是守护线程的特点，用一句话概括就是，当守护线程所守护的线程结束时，守护线程自身也会自动关闭。

线程如何退出

要知道当Thread的run方法执行结束后，线程便会自动退出，生命周期结束，这属于线程正常退出的范畴。在实际的开发过程中，还存在另一种业务场景，因为某种原因，需要停止正在运行的线程，那该怎么办呢？

在Thread中提供了一个stop方法，stop方法是JDK提供的一个可以强制关闭线程的方法，**但是不建议使用**！而且Android针对JDK中的Thread进行了修改，在Android的Thread中调用stop方法会直接抛出异常：

```
@Deprecated
public final synchronized void stop(Throwable obj) {
    throw new UnsupportedOperationException();
}
```

那么当我们需要停止运行的线程时，可以通过标识位的方式实现：

使用标志位退出线程

线程执行run方法过程中，我们可以通过一个自定义变量来决定是否还需要退出线程，若满足条件，则退出线程，反之继续执行：

下面代码：注释已经加的很详细了，直接从日志的输出结果来看，很完美的停止了子线程的运行，很OK

```
public class StopThreadTest {
    // 定义标志位，使用volatile，保证内存可见
    private static volatile boolean stopFlag = false;
    public static void main(String[] args) {
        // 创建子线程
        Thread thread = new Thread() {
            @Override
            public void run() {
                super.run();
                // 循环打印运行日志
                while (!stopFlag) {
                    System.out.println(currentThread().getName() + " is
running");
                }
                // 退出后，打印退出日志
                if (stopFlag) {
                    System.out.println(currentThread().getName() + " is stop");

                }
            }
        };
        thread.start();
        // 让子线程执行100ms后，将stopFlag置为true
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        stopFlag = true;
    }
}
```

```
// logcat日志// Thread-0 is running// Thread-0 is running// Thread-0 is running//
Thread-0 is running// Thread-0 is running// Thread-0 is running// Thread-0 is
running// Thread-0 is running// Thread-0 is running// Thread-0 is running//
Thread-0 is stop
```

使用interrupt方法中断线程

使用interrupt方法来中断线程，本质上也是标志位的方式。跟上面的原理一样，只不过是Thread类内部提供的罢了。其他线程通过调用某个线程的interrupt()方法对其进行中断操作，被中断的线程则是通过线程的isInterrupted()来进行判断是否被中断。

```
public class StopThreadTest {
    public static void main(String[] args) {
        // 创建子线程
        Thread thread = new Thread() {
            @Override
            public void run() {
                super.run();
                // 通过isInterrupted()方法判断线程是否已经中断
                while (!isInterrupted()) {
                    // 打印运行日志
                    System.out.println(currentThread().getName() + " is
running");
                }
                if (isInterrupted()) {
                    // 打印退出日志
                    System.out.println(currentThread().getName() + " is stop");

                }
            }
        };
        thread.start();
        // 让子线程执行100ms后，将stopFlag置为true
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        // 通过调用interrupt方法去改变标志位
        thread.interrupt();
    }
}
```

4.17 sleep 、wait、yield与join的区别，wait 的线程如何唤醒它？（字节跳动）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门1》

这道题想考察什么？

1. 是否了解Java中线程相关的知识点
2. 线程的生命周期

考察的知识点

1. Java中线程的相关概念
2. Sleep、yield、wait和jion函数的区别
3. 多线程并发相关的知识点

考生应该如何回答

sleep 、wait、yield与join的区别

sleep、yield与join是线程方法，而wait则是Object方法：

- sleep ，**释放cpu资源，不释放锁资源，**如果线程进入sleep的话，释放cpu资源，如果外层包有Synchronize，那么此锁并没有释放掉。
- wait，**释放cpu资源，也释放锁资源，**一般用于锁机制中 肯定是要释放掉锁的，因为notify并不会立即调起此线程，因此cpu是不会为其分配时间片的，也就是说wait 线程进入等待池，cpu不分时间片给它，锁释放掉。
- yield：**让出CPU调度**，Thread类的方法，类似sleep只是**不能由用户指定暂停多长时间 ，**并且yield()方法**只能让同优先级的线程**有执行的机会。 yield()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。调用yield方法只是一个建议，告诉线程调度器我的工作已经做的差不多了，可以让别的相同优先级的线程使用CPU了，没有任何机制保证采纳。
- join：一种特殊的wait，当前运行线程调用另一个线程的join方法，当前线程进入阻塞状态直到另一个线程运行结束等待该线程终止。 注意该方法也需要捕捉异常。

wait 的线程如何唤醒

执行wait方法会导致当前线程进入Blocked状态，可以调用执行wait方法的对象的notify或者notifyAll方法唤醒：

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        var q = new TaskQueue();
        var ts = new ArrayList<Thread>();
        for (int i=0; i<5; i++) {
            var t = new Thread() {
                public void run() {
                    // 执行task:
                    while (true) {
                        try {
                            String s = q.getTask();
                            System.out.println("execute task: " + s);
                        } catch (InterruptedException e) {
                            return;
                        }
                    }
                }
```

```
            };
            t.start();
            ts.add(t);
        }
        var add = new Thread(() -> {
            for (int i=0; i<10; i++) {
                // 放入task:
                String s = "t-" + Math.random();
                System.out.println("add task: " + s);
                q.addTask(s);
                try { Thread.sleep(100); } catch(InterruptedException e) {}
            }
        });
        add.start();
        add.join();
        Thread.sleep(100);
        for (var t : ts) {
            t.interrupt();
        }
    }
}

class TaskQueue {
    Queue<String> queue = new LinkedList<>();

    public synchronized void addTask(String s) {
        this.queue.add(s);
        this.notifyAll();
    }

    public synchronized String getTask() throws InterruptedException {
        while (queue.isEmpty()) {
            this.wait();
        }
        return queue.remove();
    }
}
```

在addTask()方法中调用了`this.notifyAll()`而不是`this.notify()`，使用notifyAll()将唤醒所有当前正在`this`锁等待的线程，而notify()只会唤醒其中一个（具体哪个依赖操作系统，有一定的随机性）。

之所以使用notifyAll，是因为可能有多个线程正在getTask()方法内部的`wait()`中等待，使用notifyAll()将一次性全部唤醒。通常来说，notifyAll()更安全。有些时候，如果我们的代码逻辑考虑不周，用notify()会导致只唤醒了一个线程，而其他线程可能永远处于Blocked状态。

4.18 sleep是可中断的么？（小米）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门1》

这道题想考察什么？

是否能够在真实场景中合理运用sleep

考察的知识点

线程管理

考生应该如何回答

sleep是可中断的。

```
/**
 * Causes the currently executing thread to sleep (temporarily cease
 * execution) for the specified number of milliseconds, subject to
 * the precision and accuracy of system timers and schedulers. The thread
 * does not lose ownership of any monitors.
 *
 * @param  millis
 *         the length of time to sleep in milliseconds
 *
 * @throws  IllegalArgumentException
 *          if the value of {@code millis} is negative
 *
 * @throws  InterruptedException
 *          if any thread has interrupted the current thread. The
 *          <i>interrupted status</i> of the current thread is
 *          cleared when this exception is thrown.
 */
// BEGIN Android-changed: Implement sleep() methods using a shared native
implementation.
public static void sleep(long millis) throws InterruptedException {
    sleep(millis, 0);
}
```

在现场中

4.19 怎么保证线程按顺序执行？如何实现线程排队 ?(金山)

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门1》、线程与进程的理论知识入门2》与《阻塞队列与线程池原理》

这道题想考察什么？

是否了解多个线程顺序启动的方式有哪些与真实场景使用，是否熟悉多个线程顺序启动在工作中的表现是什么？

考察的知识点

多个线程顺序启动的方式有哪些的概念在项目中使用与基本知识

考生应该如何回答

Q：假设有A、B两个线程，B线程需要在A线程执行完成之后执行。

A：可以在启动B线程之前，调用A线程的join方法，让B线程在A线程执行完成之后启动。

```
Thread t1 = new Thread() {
    @Override
    public void run() {
          System.out.println("执行第一个线程任务！");
    }
};
t1.start();
t1.join(); //阻塞等待线程1执行完成
Thread t2 = new Thread() {
    @Override
    public void run() {
        System.out.println("执行第二个线程任务！");
    }
};
t2.start();
```

Q: 假设有A、B两个线程，其中A线程中执行分为3步，需要在A线程执行完成第二步之后再继续执行B线程的代码怎么办？

A：可以使用wait/notify，在B线程中wait，A线程执行完成第二步之后执行notify通知B线程继续执行。

```
Object lock = new Object();
Thread t1 = new Thread() {
    @Override
    public void run() {
        System.out.println("第一步执行完成！");
        System.out.println("第二步执行完成！");
         synchronized (lock) {
            lock.notify();
         }
        System.out.println("第三步执行完成！");
    }
};

Thread t2 = new Thread() {
    @Override
    public void run() {
        synchronized (lock) {
            try {
                lock.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("执行第二个线程任务！");
```

```
    }
};
t2.start(); //注意必须先启动t2线程，否则可能t2在notify之后才wait!
t1.start();
```

Q：假设有A、B、C三个线程，其中A、B线程执行分为三步，C线程需要在A线程执行完第二步后执行一部分代码然后继续等待B线程都执行完第二步时才能执行，怎么办？

A：可以借助CountDownLatch闭锁来完成：

```
CountDownLatch countDownLatch = new CountDownLatch(2);
Thread t1 = new Thread() {
    @Override
    public void run() {
        System.out.println("t1:第一步执行完成！");
        System.out.println("t1:第二步执行完成！");
        countDownLatch.countDown();
        System.out.println("t1:第三步执行完成！");
    }
};

Thread t2 = new Thread() {
    @Override
    public void run() {
        System.out.println("t2:第一步执行完成！");
        System.out.println("t2:第二步执行完成！");
        countDownLatch.countDown();
        System.out.println("t2:第三步执行完成！");
    }
};
Thread t3 = new Thread() {
    @Override
    public void run() {
    try {
        countDownLatch.await();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("执行第三个线程任务！");
};
t3.start();
t2.start();
t1.start();
```

4.20 非阻塞式生产者消费者如何实现（字节跳动）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《阻塞队列与线程池原理》

这道题想考察什么？

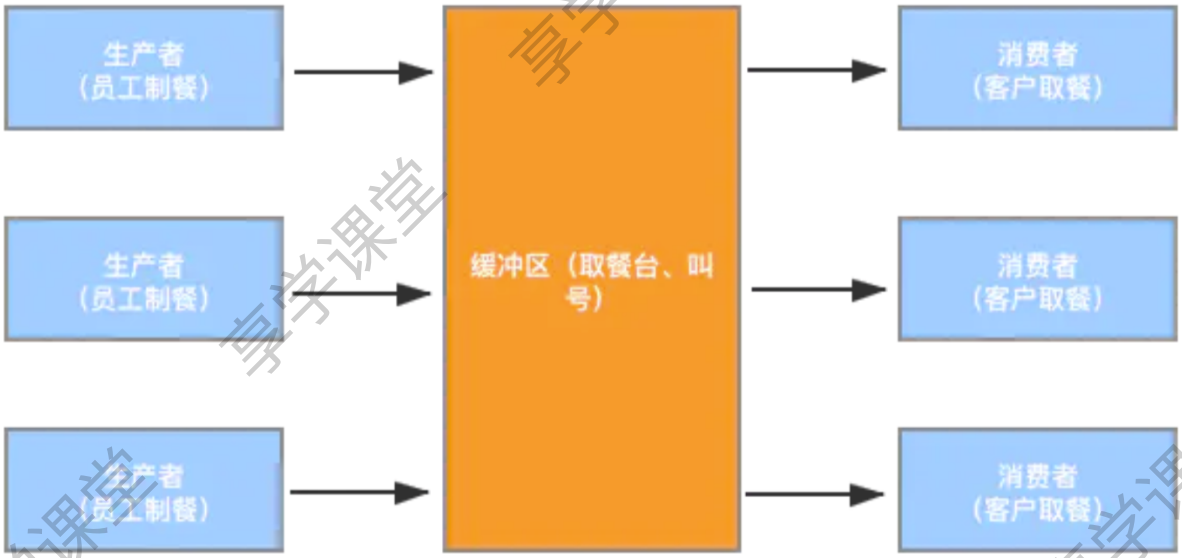
是否了解非阻塞式生产者消费者与真实场景使用，是否熟悉非阻塞式生产者消费者

考察的知识点

非阻塞式生产者消费者的概念在项目中使用与基本知识

考生应该如何回答

生产者消费者模式在日常生活中，生产者消费者模式特别常见。比如说我们去麦当劳吃饭，在前台点餐，付完钱后并不是直接给你汉堡薯条啥的，而是给你一张小票，你需要前去取餐处等待，后厨加工完的餐食都直接放入取餐处，机器叫号提醒，客户凭小票取餐。



生产者消费者模型

上面取餐的场景其实就是一个典型的生产者消费者模型，具备3个部分：生产者、消费者、缓冲区。后厨就相当于生产者，客户就是消费者，而取餐台是两者之间的一个缓冲区。再转到我们平时开发过程中，经常会碰到这样子的场景：某个模块负责产生数据，这些数据由另一个模块来负责处理。产生数据的模块，就称为生产者，而处理数据的模块，就称为消费者。当然如果只抽象出生产者和消费者，还不是正儿八经的生产者消费者模式，还需要一个缓冲区，生产者生产数据到缓冲区，消费者从缓冲区拿数据去消费。服务器端经常使用的消息队列设计就是参照生产者消费者模型。但这个时候有的同学就会好奇的问一句，干嘛需要缓冲区呢，生产完直接给消费者不是更加简单吗？在复杂的系统中，这中间的缓冲区必不可少，作用明显。

- 解耦。这是最显而易见的，如果生产者直接将数据交给消费者，那么这两个类必然会有依赖，消费者的改动都会影响到生产者。当两者之间加入缓存区之后，生产者与消费者之间彻底解耦了，各有所职，互不依赖。
- 平衡生产与消费能力。在多线程的环境下，如果生产者生产数据速度很快，消费者来不及消费，那么缓冲区便是生产者数据暂存的地方，生产者生产完一个数据后直接丢在缓冲区，便可以去生产下一个数据，消费者自己从缓冲区拿数据慢慢处理，这样生产者无需因为消费者的处理能力弱而浪费资源。当然，反之也一样。

阻塞与非堵塞

什么是堵塞？通俗的话来讲，就是一件事没干完，就只能在这等待，不允许去做其他的事。在程序世界里，阻塞调用是指调用结果返回之前，当前线程会被挂起。调用线程只有在得到结果之后才会返回。而非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程，线程让出CPU。

要求实现非堵塞式生产消费模式，所以缓冲区我们就不能使用便捷的堵塞队列，只能使用一般的集合代替，类似ArrayList等。很明显会引起两个问题。

第一，并发问题。类似ArrayList这些普通集合是线程不安全的，当生产者、消费者线程同时操作(数据入队、数据出队)缓冲区时，必然会引起ConcurrentModificationException。当然这个问题解决方案有很多种，比如使用synchronized、ReentrantLock等锁机制，也可以使用线程安全的集合，当然线程安全的集合底层也是锁机制。

第二，线程通信问题。非堵塞式的意思就是生产者与消费者去操作缓冲区，只是尝试去操作，至于能不能得到想要的结果，他们是不管的，并不会像堵塞队列那样死等。那么生产者与消费者之间的协作与通信，比如缓冲区没数据时通知生产者去生产；缓冲区有数据后，通知消费者去消费；当缓冲区数据满了让生产者休息。这里我们可以使用wait、notify/notifyAll方法。这些方法使用过程中注意以下几点基本就可以了。

- wait() 和 notify() 使用的前提是必须先获得锁，一般配合synchronized关键字使用，即在synchronized同步代码块里使用 wait()、notify/notifyAll() 方法。
- 当线程执行wait()方法时候，会释放当前持有的锁，然后让出CPU，当前线程进入等待状态。
- 当notify()方法执行时候，会唤醒正处于等待状态的线程，使其继续执行，notify()方法不会立即释放锁，锁的释放要看同步代码块的具体执行情况。notifyAll()方法的功能也是类似。
- notify()方法只唤醒一个等待线程并使该线程开始执行。所以如果有多个线程等待，这个方法只会唤醒其中一个线程，选择哪个线程取决于操作系统对多线程管理的实现。notifyAll()方法会唤醒所有等待线程，至于哪一个线程将会第一个处理取决于操作系统的实现。

经过分析，非堵塞式生产者消费者模式实现为：

```
/**
 * 实现非堵塞式生产者消费者模式
 */
public class ProducerConsumerDemo {

    /**
     * 定义队列最大容量，指缓冲区最多存放的数量
     */
    private static int MAX_SIZE = 3;

    /**
     * 缓冲区队列，ArrayList为非堵塞队列，线程不安全
     * static修饰，全局唯一
     */
    private static final List<String> list = new ArrayList<>();

    public static void main(String[] args) {
        //创建生产者线程
        Producer producer = new Producer();
        //创建消费者线程
        Consumer consumer = new Consumer();
        //生产者线程开启
        producer.start();
        //消费者线程开启
        consumer.start();
    }
```

```
    /**
     * 生产者线程
     */
    static class Producer extends Thread {

        @Override
        public void run() {
            //具体实现...
        }
    }

    /**
     * 消费者线程
     */
    static class Consumer extends Thread {

        @Override
        public void run() {
            //具体实现...
        }
    }
}
```

类的大致结构如上所示，很简单也很清晰。我们实现了两个线程，一个代表生产者，一个代表消费者，缓冲区使用非阻塞式队列ArrayList，所以它是线程不安全的，为了能更加清晰的看出生产者消费者执行流程，缓冲区大小设置成较小的3。

生产者线程

```
/**
 * 生产者线程
 */
static class Producer extends Thread {

        @Override
        public void run() {
            //使用while循环执行run方法
            while (true) {
                try {
                    //生产者 sleep 300ms，消费者 sleep 500ms，模拟两者的处理能力不均衡
                    Thread.sleep(300);
                } catch (InterruptedException e1) {
                    e1.printStackTrace();
                }

                //第1步：获取队列对象的锁，与消费者持有的锁是同一把，保证线程安全
                synchronized (list) {
                    //第2步：判断缓冲区当前容量
                    //第2.1步：队列满了就不生产，等待
                    while (list.size() == MAX_SIZE) {
                        System.out.println("生产者 -> 缓冲区满了，等待消费...");
                        try {
                            //使用wait等待方法，内部会释放当前持有的锁
                            list.wait();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
```

```
                    }
                }
                //第2.2步：队列未满就生产一个产品
                list.add("产品");
                System.out.println("生产者 -> 生产一个产品，当前队列大小：" +
list.size());

                //唤醒其他线程，这里其他线程就是指消费者线程
                list.notify();
            }
        }
    }
}
```

生产者线程负责生产数据。只要开始执行生产流程，第一步先获取list对象锁，也就意味着当前只有生产者线程可操作缓冲区，保证线程安全。第二步它会先检查一下当前缓存区的容量，如果缓存区已经满了，那生产者无需再去生产新的数据，调用wait方法进行等待，这个过程会释放list对象锁。如果缓冲区没满，就直接生产一个产品，并通过notify方法唤醒消费者线程。

消费者线程

```
/**
 * 消费者线程
 */
static class Consumer extends Thread {

    @Override
    public void run() {
        //使用while循环执行run方法
        while (true) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException e1) {
                e1.printStackTrace();
            }

            //第1步：获取队列对象的锁，与生产者持有的锁是同一把，保证线程安全
            synchronized (list) {
                //第2步：判断缓冲区当前容量
                  //第2.1步：队列空了，等带
                while (list.size() == 0) {
                    System.out.println("消费者 -> 缓冲区空了，等待生产...");
                    try {
                        //使用wait等待方法，内部会释放当前持有的锁
                        list.wait();
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
                //第2.2步：队列不为空，消费一个产品
                list.remove(0);
                System.out.println("消费者 -> 消费一个产品，当前队列大小：" +
list.size());

                //唤醒其他线程，这里其他线程就是指生产者线程
                list.notify();
            }
```

```
            }
        }
    }
```

消费者线程负责消费数据，它的实现与生产者相似。第一步也是先获取list对象锁，避免并发异常。第二步检查当前缓冲区容量时，这里与生产者正好相反。如果缓冲区已经空了，没有数据可消费了，它会使用wait方法进行等待。如果缓冲区没空，则去消费一个产品，并且调用notify方法唤醒生产者线程去生产。

执行ProducerConsumerDemo的main方法，抓取方法执行日志。从logcat日志可以看出，生产者与消费者互相协作，有条不紊的进行生产与消费操作，没有引起并发异常问题。

```
//logcat日志（截取了部分）
生产者 -> 生产一个产品，当前队列大小：1
消费者 -> 消费一个产品，当前队列大小：0
生产者 -> 生产一个产品，当前队列大小：1
生产者 -> 生产一个产品，当前队列大小：2
消费者 -> 消费一个产品，当前队列大小：1
生产者 -> 生产一个产品，当前队列大小：2
消费者 -> 消费一个产品，当前队列大小：1
生产者 -> 生产一个产品，当前队列大小：2
生产者 -> 生产一个产品，当前队列大小：3
消费者 -> 消费一个产品，当前队列大小：2
生产者 -> 生产一个产品，当前队列大小：3
生产者 -> 缓冲区满了，等待消费...
消费者 -> 消费一个产品，当前队列大小：2
生产者 -> 生产一个产品，当前队列大小：3
生产者 -> 缓冲区满了，等待消费...
消费者 -> 消费一个产品，当前队列大小：2
生产者 -> 生产一个产品，当前队列大小：3
生产者 -> 缓冲区满了，等待消费...
消费者 -> 消费一个产品，当前队列大小：2
生产者 -> 生产一个产品，当前队列大小：3
生产者 -> 缓冲区满了，等待消费...
消费者 -> 消费一个产品，当前队列大小：2
...
```

4.21 线程池管理线程原理

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《阻塞队列与线程池原理》

这道题想考察什么？

是否了解线程池相关的理论知识

考察的知识点

1. 线程中的基本概念，线程的生命周期
2. 线程池的原理
3. 常见的几种线程池的特点以及各自的应用场景

考生应该如何回答

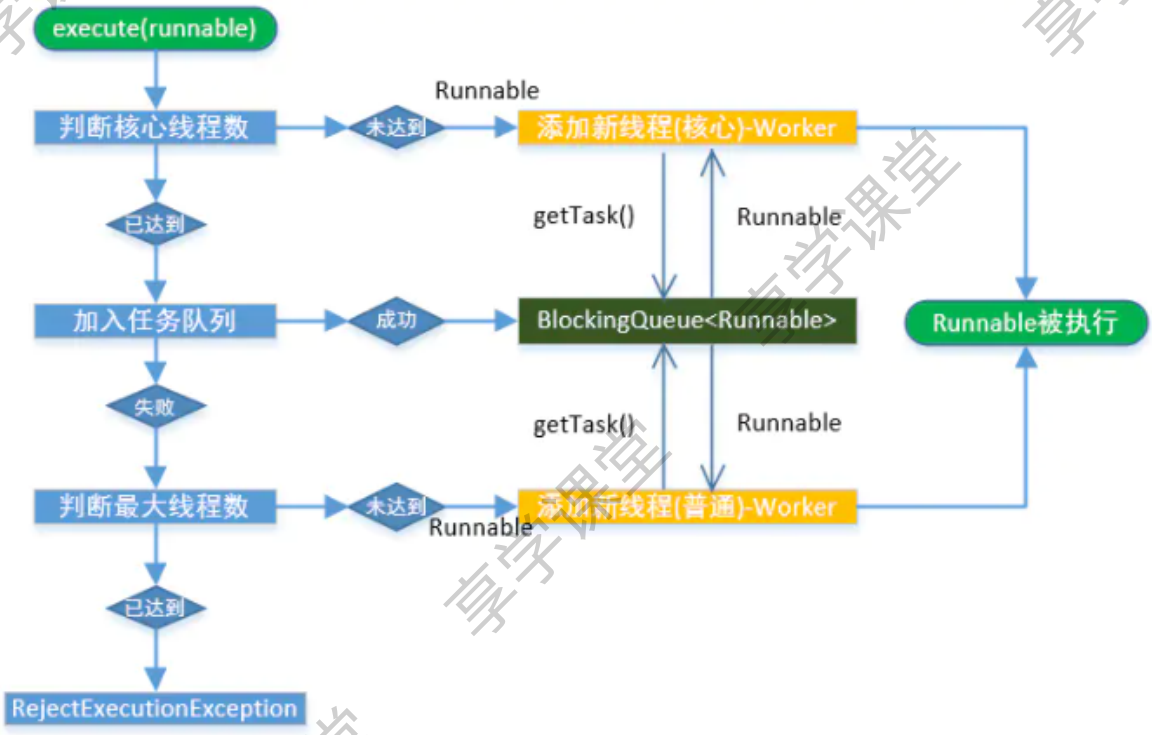
在一个应用程序中，我们需要多次使用线程，也就意味着，我们需要多次创建并销毁线程。而创建并销毁线程的过程势必会消耗内存。而在Java中，内存资源是及其宝贵的，所以，我们就提出了线程池的概念。线程池的好处，就是可以方便的管理线程，也可以减少内存的消耗。

线程池的创建

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
```

1. corePoolSize ：核心线程数量，默认情况下，线程池会一直维护corePoolSize个线程，让这些线程不会被回收。
2. maximumPoolSize：线程池最大线程的数量；
3. keepAliveTime：线程的最长闲置时间，若线程闲置超过此时间则回收；
4. util：闲置时间单位；
5. workQueue：等待队列，当线程池中执行的任务超过核心线程数后，新提交任务将加入此队列等待执行；
6. threadFactory：创建线程的线程工厂
7. handler：拒绝策略，在任务满了之后，如何处理继续添加的任务。

线程池的执行流程



在新建的线程池中，默认最开始里面是没有线程的。当然，可以使用 `prestartAllCoreThreads` 方法，来提前把corePoolSize的核心线程。

向线程池中提交任务时，首先判断线程池中正在执行的线程数是否已经达到核心线程数：

- 未达到：创建新线程执行任务

- 达到：将任务添加进入任务队列

在向任务队列添加任务时，可能添加成功，也可能添加失败：

- 成功：等待其他线程执行完成，将会自动从队列中获取任务继续执行；
- 失败：判断当前线程池线程数是否达到最大线程数

若添加队列失败，在判断是否达到最大线程数是也可能存在两种结果：

- 达到：回调`RejectedExecutionHandler`拒绝策略，JDK提供了四种拒绝策略处理类：AbortPolicy（抛出一个异常，默认的），DiscardPolicy（直接丢弃任务），DiscardOldestPolicy（丢弃队列里最老的任务，将当前这个任务继续提交给线程池），CallerRunsPolicy（交给线程池调用所在的线程进行处理）
- 未达到：创建新线程执行任务

核心线程

默认情况下，线程池会一直维护核心线程，使其不被回收。

> 可以使用`allowCoreThreadTimeOut(true)`设置核心线程也会被闲置回收。

线程池运行线程执行任务：

```
while (task != null || (task = getTask()) != null) {
    //......
    task.run();
}
```

其中`getTask`则会从队列中获取待执行任务，不断执行。如果当前队列中没有待执行任务，非核心线程会等待指定的keepAliveTime时间到达后正常退出线程，而核心线程会通过任务队列的`take`方法阻塞，从而保证其持续运行。

```
Runnable r = timed ?
                    //非核心线程
                    workQueue.poll(keepAliveTime, TimeUnit.NANOSECONDS) :
                    //核心线程
                    workQueue.take();
```

4.22 线程池有几种实现方式，线程池的七大参数有哪些？(美团)

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《阻塞队列与线程池原理》

这道题想考察什么？

是否了解线程池的七大参数有哪些与真实场景使用，是否熟悉线程池的七大参数有哪些

考察的知识点

线程池的七大参数有哪些的概念在项目中使用与基本知识

考生应该如何回答

七大参数

1. corePoolSize

   线程池中的常驻核心线程数

2. maximumPoolSize

   线程池能够容纳同时执行的最大线程数，此值必须大于等于1

3. keepAliveTime

   空闲线程的存活时间。

4. unit

   keepAliveTime的单位

5. workQueue

   任务队列，被提交但尚未被执行的任务

6. threadFactory

   表示生成线程池中工作线程的线程工厂，用于创建线程一般默认即可

7. handler：

   拒绝策略，表示当队列满了并且工作线程大于等于线程池最大线程数（maximumPoolSize）时如何处理

实现方式

使用JDK中自带的线程池可以通过创建ThreadPoolExecutor线程池对象，也能够通过Executors中定义的静态方法。其中Executors静态方法创建的线程池主要有以下类型：

1.newSingleThreadExecutor

创建只有一个线程的线程池，且线程的存活时间是无限的；当该线程正繁忙时，对于新任务会进入阻塞队列中(无界的阻塞队列)

适用：一个任务一个任务执行的场景

2.newCachedThreadPool

当有新任务到来，则插入到SynchronousQueue中，由于SynchronousQueue是同步队列，因此会在池中寻找可用线程来执行，若有可以线程则执行，若没有可用线程则创建一个线程来执行该任务；若池中线程空闲时间超过指定大小，则该线程会被销毁。

适用:执行很多短期异步的场景

3.newFixedThreadPool

创建可容纳固定数量线程的池子，每隔线程的存活时间是无限的，当池子满了就不在添加线程了；如果池中的所有线程均在繁忙状态，对于新任务会进入阻塞队列中(无界的阻塞队列)，但是，在线程池空闲时，即线程池中没有可运行任务时，它不会释放工作线程，还会占用一定的系统资源。

适用:长期执行的场景

4.NewScheduledThreadPool

创建一个固定大小的线程池，线程池内线程存活时间无限制，线程池可以支持定时及周期性任务执行，如果所有线程均处于繁忙状态，对于新任务会进入DelayedWorkQueue队列中，这是一种按照超时时间排序的队列结构

适用:周期性执行的场景

4.23 如何开启一个线程，开启大量线程会有什么问题，如何优化？（美团）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门1》

这道题想考察什么？

是否了解开启大量线程会有什么问题与真实场景使用，是否熟悉开启大量线程会有什么问题？

考察的知识点

开启大量线程会有什么问题的概念在项目中使用与基本知识

考生应该如何回答

如何开启一个线程

如何开启一个线程，再JDK中的说明为：

```
/**
 * ...
 * There are two ways to create a new thread of execution. One is to
 * declare a class to be a subclass of <code>Thread</code>.
 * The other way to create a thread is to declare a class that
 * implements the <code>Runnable</code> interface.
 * ....
 */
public class Thread implements Runnable{

}
```

Thread源码的类描述中有这样一段，翻译一下，只有两种方法去创建一个执行线程，一种是声明一个Thread的子类，另一种是创建一个类去实现Runnable接口。

继承Thread类

```
public class ThreadUnitTest {

    @Test
    public void testThread() {
        //创建MyThread实例
        MyThread myThread = new MyThread();
        //调用线程start的方法，进入可执行状态
        myThread.start();
    }

    //继承Thread类，重写内部run方法
```

```
    static class MyThread extends Thread {

        @Override
        public void run() {
            System.out.println("test MyThread run");
        }
    }
}
```

实现Runnable接口

```
public class ThreadUnitTest {

    @Test
    public void testRunnable() {
        //创建MyRunnable实例，这其实只是一个任务，并不是线程
        MyRunnable myRunnable = new MyRunnable();
        //交给线程去执行
        new Thread(myRunnable).start();
    }

    //实现Runnable接口，并实现内部run方法
    static class MyRunnable implements Runnable {

        @Override
        public void run() {
            System.out.println("test MyRunnable run");
        }
    }
}
```

实现Callable

其实实现Callback接口创建线程的方式，归根到底就是Runnable方式，只不过它是在Runnable的基础上又增加了一些能力，例如取消任务执行等。

```
public class ThreadUnitTest {

    @Test
    public void testCallable() {
        //创建MyCallable实例，需要与FutureTask结合使用
        MyCallable myCallable = new MyCallable();
        //创建FutureTask，与Runnable一样，也只能算是个任务
        FutureTask<String> futureTask = new FutureTask<>(myCallable);
        //交给线程去执行
        new Thread(futureTask).start();

        try {
            //get方法获取任务返回值，该方法是阻塞的
            String result = futureTask.get();
            System.out.println(result);
        } catch (ExecutionException e) {
            e.printStackTrace();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
    //实现Callable接口，并实现call方法，不同之处是该方法有返回值
    static class MyCallable implements Callable<String> {

        @Override
        public String call() throws Exception {
            Thread.sleep(10000);
            return "test MyCallable run";
        }
    }
}
```

Callable的方式必须与FutureTask结合使用，我们看看FutureTask的继承关系:

```
//FutureTask实现了RunnableFuture接口
public class FutureTask<V> implements RunnableFuture<V> {

}

//RunnableFuture接口继承Runnable和Future接口
public interface RunnableFuture<V> extends Runnable, Future<V> {
    void run();
}
```

开启大量线程会引起什么问题

在Java中，调用Thread的start方法后，该线程即置为就绪状态，等待CPU的调度。这个流程里有两个关注点需要去理解。

start内部怎样开启线程的？看看start方法是怎么实现的。

```
// Thread类的start方法
public synchronized void start() {
        // 一系列状态检查
        if (threadStatus != 0)
            throw new IllegalThreadStateException();

        group.add(this);

        boolean started = false;
        try {
             //调用start0方法，真正启动java线程的地方
            start0();
            started = true;
        } finally {
            try {
                if (!started) {
                    group.threadStartFailed(this);
                }
            } catch (Throwable ignore) {
            }
        }
    }
```

```
//start0方法是一个native方法
private native void start0();
```

JVM中，native方法与java方法存在一个映射关系，Java中的start0对应c层的JVM_StartThread方法，我们继续看一下:

```
JVM_ENTRY(void, JVM_StartThread(JNIEnv* env, jobject jthread))
  JVMWrapper("JVM_StartThread");
  JavaThread *native_thread = NULL;
  bool throw_illegal_thread_state = false;
  {

    MutexLocker mu(Threads_lock);
    // 判断Java线程是否已经启动，如果已经启动过，则会抛异常。
    if (java_lang_Thread::thread(JNIHandles::resolve_non_null(jthread)) != NULL)
{
      throw_illegal_thread_state = true;
    } else {
      //如果没有启动过，走到这里else分支，去创建线程
      //分配c++线程结构并创建native线程
      jlong size =
             java_lang_Thread::stackSize(JNIHandles::resolve_non_null(jthread));

      size_t sz = size > 0 ? (size_t) size : 0;
      //注意这里new JavaThread
      native_thread = new JavaThread(&thread_entry, sz);
      if (native_thread->osthread() != NULL) {
        native_thread->prepare(jthread);
      }
    }
  }
  ......
  Thread::start(native_thread);
```

走到这里发现，Java层已经过渡到native层，但远远还没结束:

```
JavaThread::JavaThread(ThreadFunction entry_point, size_t stack_sz) :
                        Thread()
  {
    initialize();
    _jni_attach_state = _not_attaching_via_jni;
    set_entry_point(entry_point);
    os::ThreadType thr_type = os::java_thread;
    thr_type = entry_point == &compiler_thread_entry ? os::compiler_thread :
                                                       os::java_thread;
    //根据平台，调用create_thread，创建真正的内核线程
    os::create_thread(this, thr_type, stack_sz);
  }

  bool os::create_thread(Thread* thread, ThreadType thr_type,
                         size_t req_stack_size) {
      ......
      pthread_t tid;
      //利用pthread_create（）来创建线程
```

```
        int ret = pthread_create(&tid, &attr, (void* (*)(void*))
thread_native_entry, thread);
        ......
        return true;
}
```

pthread_create方法，第三个参数表示启动这个线程后要执行的方法的入口，第四个参数表示要给这个方法传入的参数:

```
static void *thread_native_entry(Thread *thread) {
  ......
  //thread_native_entry方法的最下面的run方法，这个thread就是上面传递下来的参数，也就是
JavaThread
  thread->run();
  ......
  return 0;
}
```

终于开始执行run方法了:

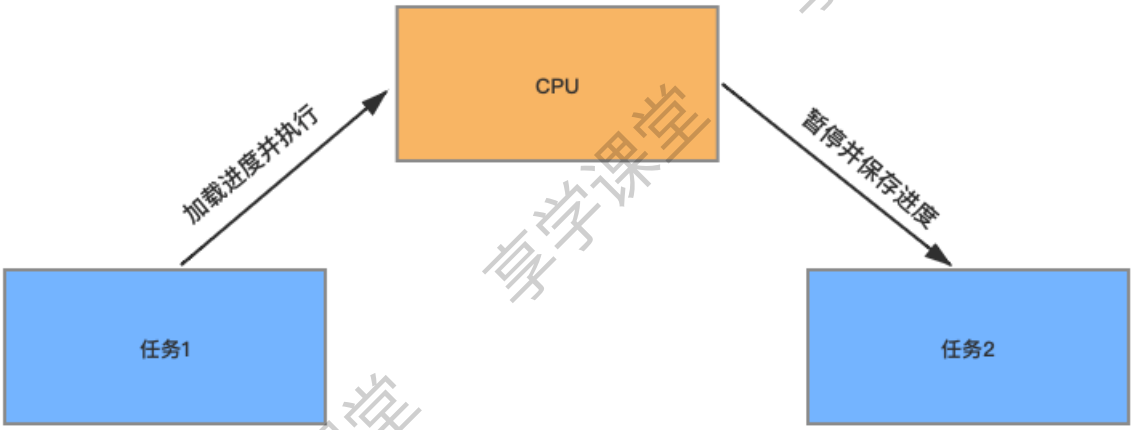
```
//thread.cpp类
void JavaThread::run() {
  ......
  //调用内部thread_main_inner
  thread_main_inner();
}

void JavaThread::thread_main_inner() {
  if (!this->has_pending_exception() &&
   !java_lang_Thread::is_stillborn(this->threadObj())) {
    {
      ResourceMark rm(this);
      this->set_native_thread_name(this->get_thread_name());
    }
    HandleMark hm(this);
    //注意：内部通过JavaCalls模块，调用了Java线程要执行的run方法
    this->entry_point()(this, this);
  }
  DTRACE_THREAD_PROBE(stop, this);
  this->exit(false);
  delete this;
}
```

一条U字型代码调用链至此结束：

- Java中调用Thread的star方法，通过JNI方式，调用到native层。
- native层，JVM通过pthread_create方法创建一个系统内核线程，并指定内核线程的初始运行地址，即一个方法指针。
- 在内核线程的初始运行方法中，利用JavaCalls模块，回调到java线程的run方法，开始java级别的线程执行。

线程如何调度

计算机的世界里，CPU会分为若干时间片，通过各种算法分配时间片来执行任务，有耳熟能详时间片轮转调度算法、短进程优先算法、优先级算法等。当一个任务的时间片用完，就会切换到另一个任务。在切换之前会保存上一个任务的状态，当下次再切换到该任务，就会加载这个状态，这就是所谓的线程的上下文切换。很明显，上下文的切换是有开销的，包括很多方面，操作系统保存和恢复上下文的开销、线程调度器调度线程的开销和高速缓存重新加载的开销等。



经过上面两个理论基础的回顾，开启大量线程引起的问题，总结起来，就两个字——开销。

消耗时间：线程的创建和销毁都需要时间，当数量太大的时候，会影响效率。
消耗内存：创建更多的线程会消耗更多的内存，这是毋庸置疑的。线程频繁创建与销毁，还有可能引起内存抖动，频繁触发GC，最直接的表现就是卡顿。长而久之，内存资源占用过多或者内存碎片过多，系统甚至会出现OOM。
消耗CPU。在操作系统中，CPU都是遵循时间片轮转机制进行处理任务，线程数过多，必然会引起CPU频繁的进行线程上下文切换。这个代价是昂贵的，某些场景下甚至超过任务本身的消耗。

如何优化

线程的本质是为了执行任务，在计算机的世界里，任务分大致分为两类，CPU密集型任务和IO密集型任务。

CPU密集型任务，比如公式计算、资源解码等。这类任务要进行大量的计算，全都依赖CPU的运算能力，持久消耗CPU资源。所以针对这类任务，其实不应该开启大量线程。因为线程越多，花在线程切换的时间就越多，CPU执行效率就越低，一般CPU密集型任务同时进行的数量等于CPU的核心数，最多再加个1。
IO密集型任务，比如网络读写、文件读写等。这类任务不需要消耗太多的CPU资源，绝大部分时间是在IO操作上。所以针对这类任务，可以开启大量线程去提高CPU的执行效率，一般IO密集型任务同时进行的数量等于CPU的核心数的两倍。
另外，在无法避免，必须要开启大量线程的情况下，我们也可以使用**线程池**代替直接创建线程的做法进行优化。线程池的基本作用就是复用已有的线程，从而减少线程的创建，降低开销。在Java中，线程池的使用还是非常方便的，JDK中提供了现成的ThreadPoolExecutor类，我们只需要按照自己的需求进行相应的参数配置即可，这里提供一个示例。

```
/**
 * 线程池使用
 */
public class ThreadPoolService {

    /**
     * 线程池变量
     */
    private ThreadPoolExecutor mThreadPoolExecutor;
```

```
    private static volatile ThreadPoolService sInstance = null;

    /**
     * 线程池中的核心线程数，默认情况下，核心线程一直存活在线程池中，即便他们在线程池中处于闲置状态。
     * 除非我们将ThreadPoolExecutor的allowCoreThreadTimeOut属性设为true的时候，这时候处于闲置的核心          * 线程在等待新任务到来时会有超时策略，这个超时时间由keepAliveTime来指定。一旦超过所设置的超时时间，闲       * 置的核心线程就会被终止。
     * CPU密集型任务  N+1   IO密集型任务   2*N
     */
    private final int CORE_POOL_SIZE =
Runtime.getRuntime().availableProcessors() + 1;
    /**
     * 线程池中所容纳的最大线程数，如果活动的线程达到这个数值以后，后续的新任务将会被阻塞。包含核心线程数+非*       * 核心线程数。
     */
    private final int MAXIMUM_POOL_SIZE = Math.max(CORE_POOL_SIZE, 10);
    /**
     * 非核心线程闲置时的超时时长，对于非核心线程，闲置时间超过这个时间，非核心线程就会被回收。
     * 只有对ThreadPoolExecutor的allowCoreThreadTimeOut属性设为true的时候，这个超时时间才会对核心线       * 程产生效果。
     */
    private final long KEEP_ALIVE_TIME = 2;
    /**
     * 用于指定keepAliveTime参数的时间单位。
     */
    private final TimeUnit UNIT = TimeUnit.SECONDS;
    /**
     * 线程池中保存等待执行的任务的阻塞队列
     * ArrayBlockingQueue   基于数组实现的有界的阻塞队列
     * LinkedBlockingQueue   基于链表实现的阻塞队列
     * SynchronousQueue    内部没有任何容量的阻塞队列。在它内部没有任何的缓存空间
     * PriorityBlockingQueue    具有优先级的无限阻塞队列。
     */
    private final BlockingQueue<Runnable> WORK_QUEUE = new LinkedBlockingDeque<>
();
    /**
     * 线程工厂，为线程池提供新线程的创建。ThreadFactory是一个接口，里面只有一个newThread方法。 默认为DefaultThreadFactory类。
     */
    private final ThreadFactory THREAD_FACTORY =
Executors.defaultThreadFactory();
    /**
     * 拒绝策略，当任务队列已满并且线程池中的活动线程已经达到所限定的最大值或者是无法成功执行任务，这时候       * ThreadPoolExecutor会调用RejectedExecutionHandler中的rejectedExecution方法。
     * CallerRunsPolicy   只用调用者所在线程来运行任务。
     * AbortPolicy   直接抛出RejectedExecutionException异常。
     * DiscardPolicy   丢弃掉该任务，不进行处理。
     * DiscardOldestPolicy    丢弃队列里最近的一个任务，并执行当前任务。
     */
    private final RejectedExecutionHandler REJECTED_HANDLER = new
ThreadPoolExecutor.AbortPolicy();

    private ThreadPoolService() {
    }
```

```
    /**
     * 单例
     * @return
     */
    public static ThreadPoolService getInstance() {
        if (sInstance == null) {
            synchronized (ThreadPoolService.class) {
                if (sInstance == null) {
                    sInstance = new ThreadPoolService();
                    sInstance.initThreadPool();
                }
            }
        }
        return sInstance;
    }

    /**
     * 初始化线程池
     */
    private void initThreadPool() {
        try {
            mThreadPoolExecutor = new ThreadPoolExecutor(
                    CORE_POOL_SIZE,
                    MAXIMUM_POOL_SIZE,
                    KEEP_ALIVE_TIME,
                    UNIT,
                    WORK_QUEUE,
                    THREAD_FACTORY,
                    REJECTED_HANDLER);
        } catch (Exception e) {
            LogUtil.printStackTrace(e);
        }
    }

    /**
     * 向线程池提交任务,无返回值
     *
     * @param runnable
     */
    public void post(Runnable runnable) {
        mThreadPoolExecutor.execute(runnable);
    }

    /**
     * 向线程池提交任务,有返回值
     *
     * @param callable
     */
    public <T> Future<T> post(Callable<T> callable) {
        RunnableFuture<T> task = new FutureTask<T>(callable);
        mThreadPoolExecutor.execute(task);
        return task;
    }
}
```

4.24 pthread 了解吗？new 一个线程占用多少内存？（快手）

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《线程与进程的理论知识入门1》

这道题想考察什么？

是否清楚创建线程的代价

考察的知识点

线程底层原理

线程资源消耗

考生应该如何回答

pthread

pthread一般是指 POSIX的线程标准，是一套定义了创建和操纵线程的API。 一般用于Unix-like系统，如Linux、Mac OS。

Java的跨平台基于虚拟机，由JVM屏蔽不同的操作系统的底层实现。在Java中创建线程，运行在Linux中(包括Android)，实际上就是封装了pthread（**《4.23 如何开启一个线程，开启大量线程会有什么问题，如何优化？》**）。

线程内存

在Java中每个线程需要分配线程内存，用来存储自身的线程变量。在JDK 1.4中每个线程是256K的内存，在JDK 1.5之后每个线程是1M的内存。

4.25 HandlerThread是什么？

详细讲解

享学课堂移动互联网系统课程：Android 底层FrameWork内核解析《Handler消息机制原理解析-中》

这道题想考察什么？

是否了解HandlerThread与真实场景使用，是否熟悉HandlerThread

考察的知识点

HandlerThread的概念在项目中使用与基本知识

考生应该如何回答

HandlerThread原理

HandlerThread就是一个自带Handler的Thread，更确切的讲是一个内含有Looper的Thread，这点从HandlerThread的源码注释就可以得知。

```
/**
 * Handy class for starting a new thread that has a looper. The looper can then
be
 * used to create handler classes. Note that start() must still be called.
 */
```

用于启动具有looper的新线程的方便类。 可以使用looper来创建处理程序类。 注意仍然必须调用start（）来启动。

要想获取一个与之绑定的Handler可以通过内置的getThreadHandler()方法获得。

```
public Handler getThreadHandler() {
    if (mHandler == null) {
        mHandler = new Handler(getLooper());
    }
    return mHandler;
}
```

那么，当从调用handlerThread.start()方法那一刻开始，其内部发生了什么呢？Thread的start()方法注释如下。

```
/**
 * Causes this thread to begin execution; the Java Virtual Machine
 * calls the <code>run</code> method of this thread.
 */
```

导致此线程开始执行; Java虚拟机调用此线程的run方法。

看看HandlerThread的run()方法。

```
@Override
public void run() {
    mTid = Process.myTid();
    Looper.prepare();
    synchronized (this) {
        mLooper = Looper.myLooper();
        notifyAll();
    }
    Process.setThreadPriority(mPriority);
    onLooperPrepared();
    Looper.loop();
    mTid = -1;
}
```

run()方法里没有什么陌生的东西，先是调用了Looper.prepare()方法创建一个Looper，然后加了一个同步锁获取当前线程的Looper对象赋值到mLooper上并唤醒所有线程，再然后设置当前线程优先级，再然后调用了onLooperPrepared()方法，最后调用Looper.loop()开启循环。

可能会有些好奇为什么在赋值mLooper时要加同步锁，后面会有解释。

看到这里，你可能会想，这跟我们自己继承Thread在run方法里开启一个Looper没什么两样吗，的确，但是官方也说了HandlerThread是一个方便类，方便我们在子线程使用Handler。

对了，上面run()方法里的onLooperPrepared()方法是干什么的？

```
/**
 * Call back method that can be explicitly overridden if needed to execute some
 * setup before Looper loops.
 */
protected void onLooperPrepared() {
}
```

如果需要在Looper循环之前执行某些设置，可以复写此方法。

emmm，那run()方法里加同步锁是为什么呢？

如果你不想通过HandlerThread为我们提供的getThreadHandler()方法来获取一个Handler，HandlerThread也提供了getLooper()方法为你提供Looper对象创建实现自己的Handler。

```
    public Looper getLooper() {
        if (!isAlive()) {
            return null;
        }

        // If the thread has been started, wait until the looper has been
created.
        synchronized (this) {
            while (isAlive() && mLooper == null) {
                try {
                    wait();
                } catch (InterruptedException e) {
                }
            }
        }
        return mLooper;
    }
```

这下就明白为啥在run()方法同步锁里赋值mLooper完成后要在再唤醒所有线程了。因为此方法将阻塞线程，直到looper已初始化。

整个HandlerThread类也就不到200行代码，除了上面那些，它还封装了线程不使用时退出Looper的方法。

```
    public boolean quit() {
        Looper looper = getLooper();
        if (looper != null) {
            looper.quit();
            return true;
        }
        return false;
    }
```

```
    public boolean quitSafely() {
        Looper looper = getLooper();
        if (looper != null) {
            looper.quitSafely();
            return true;
        }
        return false;
    }
```

可以看出，HandlerThread真的是一个方便类啊，很方便。

使用场景

相比于普通的Thread，我为什么要使用HandlerThread？

对于普通的Thread，执行一遍就结束了，一个实例不能start多次，如果你不止一次启动同一个示例，那么将会抛出一个IllegalThreadStateException异常，这些在Thread的源码里都有注释写到。

```
/**
 * It is never legal to start a thread more than once.
 * 不止一次启动线程永远不合法
 * In particular, a thread may not be restarted once it has completed execution.
 * 特别是，一旦完成执行，线程可能无法重新启动。
 * @exception  IllegalThreadStateException  if the thread was already started.

 *             如果线程已经启动将抛出IllegalThreadStateException异常
 */
```

如果需要频繁的操作数据库、大文件、请求网络，就需要频繁的创建Thread示例调用start()，又或是需要执行一系列串行操作；而HandlerThread由于自带Handler，也就相当自带了一个任务队列，需要进行耗时操作的时候只要通过Handler执行send或post系列操作就行，不再需要频繁创建Thread，节省资源。

4.26 AsyncTask的原理

这道题想考察什么？

是否了解AsyncTask的原理与真实场景使用，是否熟悉AsyncTask的原理

考察的知识点

AsyncTask的原理的概念在项目中使用与基本知识

考生应该如何回答

AsyncTask是Android给我们提供的一个轻量级的用于处理异步任务的类。

构造函数

```
/**
     * Creates a new asynchronous task. This constructor must be invoked on the
UI thread.
     */
    public AsyncTask() {
```

```
        mWorker = new WorkerRunnable<Params, Result>() {
            public Result call() throws Exception {
                mTaskInvoked.set(true);
                Result result = null;
                try {

Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND);
                    //noinspection unchecked
                    result = doInBackground(mParams);
                    Binder.flushPendingCommands();
                } catch (Throwable tr) {
                    mCancelled.set(true);
                    throw tr;
                } finally {
                    postResult(result);
                }
                return result;
            }
        };

        mFuture = new FutureTask<Result>(mWorker) {
            @Override
            protected void done() {
                try {
                    postResultIfNotInvoked(get());
                } catch (InterruptedException e) {
                    android.util.Log.w(LOG_TAG, e);
                } catch (ExecutionException e) {
                    throw new RuntimeException("An error occurred while
executing doInBackground()",
                            e.getCause());
                } catch (CancellationException e) {
                    postResultIfNotInvoked(null);
                }
            }
        };
    }
```

初始化了两个变量，mWorker和mFuture，并在初始化mFuture的时候将mWorker作为参数传入。mWorker是一个Callable对象，mFuture是一个FutureTask对象，这两个变量会暂时保存在内存中，稍后才会用到它们。 FutureTask实现了Runnable接口。

mWorker中的call()方法执行了耗时操作，即`result = doInBackground(mParams);`,然后把执行得到的结果通过`postResult(result);`,传递给内部的Handler跳转到主线程中。在这里这是实例化了两个变量，并没有开启执行任务。

execute方法

```
    @MainThread
    public final AsyncTask<Params, Progress, Result> execute(Params... params) {
        return executeOnExecutor(sDefaultExecutor, params);
    }
```

`execute`方法调用了`executeOnExecutor`方法并传递参数sDefaultExecutor和params。
再看`executeOnExecutor`方法：

```
public final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor
exec,
            Params... params) {

        //判断当前状态
        if (mStatus != Status.PENDING) {
            switch (mStatus) {
                case RUNNING:
                    throw new IllegalStateException("Cannot execute task:"
                            + " the task is already running.");
                case FINISHED:
                    throw new IllegalStateException("Cannot execute task:"
                            + " the task has already been executed "
                            + "(a task can be executed only once)");
            }
        }

        //将状态置为运行态
        mStatus = Status.RUNNING;

        //主线程中最先调用onPreExecute方法，进行准备工作
        onPreExecute();

        //将参数传给mWorker
        mWorker.mParams = params;

        //调用线程池，执行任务
        exec.execute(mFuture);

        return this;
    }
```

`executeOnExecutor`方法首先判断状态，若处于可执行态，则将状态置为RUNNING。然后调用了`onPreExecute`方法，交给用户进行执行任务前的准备工作。核心部分在于`exec.execute(mFuture)`。exec即sDefaultExecutor。

线程池

查看sDefaultExecutor定义：

```
public static final Executor SERIAL_EXECUTOR = new SerialExecutor();
private static volatile Executor sDefaultExecutor = SERIAL_EXECUTOR;
```

再看一下SerialExecutor类

```
    private static class SerialExecutor implements Executor {
        final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
        Runnable mActive;

        public synchronized void execute(final Runnable r) {
            mTasks.offer(new Runnable() {
                public void run() {
                    try {
                        r.run();
                    } finally {
                        scheduleNext();
```

```
                    }
                }
            });
            if (mActive == null) {
                scheduleNext();
            }
        }

        protected synchronized void scheduleNext() {
            if ((mActive = mTasks.poll()) != null) {
                THREAD_POOL_EXECUTOR.execute(mActive);
            }
        }
    }
```

sDefaultExecutor是一个串行线程池，作用在于任务的排队执行。

从SerialExecutor的源码可以看出，mFuture是插入到mTasks任务队列的对象。当mTasks中没有正在活动的AsyncTask任务，则调用`scheduleNext`方法执行下一个任务。若一个AsyncTask任务执行完毕，则继续执行下一个AsyncTask任务，直至所有任务执行完毕。通过分析可以发现真正去执行后台任务的是线程池THREAD_POOL_EXECUTOR。

在这个方法中，有两个主要步骤。

1. 向队列中加入一个新的任务，即之前实例化后的mFuture对象。
2. 调用`scheduleNext()`方法，调用THREAD_POOL_EXECUTOR执行队列头部的任务。

THREAD_POOL_EXECUTOR定义如下：

```
    /**
     * An {@link Executor} that can be used to execute tasks in parallel.
     */
    public static final Executor THREAD_POOL_EXECUTOR;

    static {
        ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(
                CORE_POOL_SIZE, MAXIMUM_POOL_SIZE, KEEP_ALIVE_SECONDS,
TimeUnit.SECONDS,
                sPoolWorkQueue, sThreadFactory);
        threadPoolExecutor.allowCoreThreadTimeOut(true);
        THREAD_POOL_EXECUTOR = threadPoolExecutor;
    }
```

实际是个线程池，开启了一定数量的核心线程和工作线程。然后调用线程池的execute()方法。执行具体的耗时任务，即开头构造函数中mWorker中call()方法的内容。先执行完doInBackground()方法，又执行postResult()方法，下面看该方法的具体内容：

```
    private Result postResult(Result result) {
        @SuppressWarnings("unchecked")
        Message message = getHandler().obtainMessage(MESSAGE_POST_RESULT,
                new AsyncTaskResult<Result>(this, result));
        message.sendToTarget();
        return result;
    }
```

该方法向Handler对象发送了一个消息，下面具体看AsyncTask中实例化的Hanlder对象—InternalHandler的源码：

```
    private static class InternalHandler extends Handler {
        public InternalHandler() {
            super(Looper.getMainLooper());
        }

        @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
        @Override
        public void handleMessage(Message msg) {
            AsyncTaskResult<?> result = (AsyncTaskResult<?>) msg.obj;
            switch (msg.what) {
                case MESSAGE_POST_RESULT:
                    // There is only one result
                    result.mTask.finish(result.mData[0]);
                    break;
                case MESSAGE_POST_PROGRESS:
                    result.mTask.onProgressUpdate(result.mData);
                    break;
            }
        }
    }
```

InternalHandler接收到MESSAGE_POST_RESULT时调用`result.mTask.finish(result.mData[0])`;即执行完了doInBackground()方法并传递结果，那么就调用finish()方法。

```
    private void finish(Result result) {
        if (isCancelled()) {
            onCancelled(result);
        } else {
            onPostExecute(result);
        }
        mStatus = Status.FINISHED;
    }
```

`finish`方法中若任务没有取消则调用`onPostExecute`方法发送结果，若任务取消则调用`onCancelled`方法。finish方法是在主线程中执行的。

InternalHandler是一个静态类，为了能够将执行环境切换到主线程，因此这个类必须在主线程中进行加载。所以变相要求AsyncTask的类必须在主线程中进行加载。

通过上述流程已经顺序找到了`onPreExecute`、`doInBackground`、`onPostExecute`方法，那么`onProgressUpdate`是如何执行的呢？
首先查看`publishProgress`方法：

```
    protected final void publishProgress(Progress... values) {
        if (!isCancelled()) {
            getHandler().obtainMessage(MESSAGE_POST_PROGRESS,
                    new AsyncTaskResult<Progress>(this, values)).sendToTarget();
        }
    }
```

在`doInBackground`中调用`publishProgress`方法，`publishProgress`方法发送MESSAGE_POST_PROGRESS消息和进度values，InternalHandler在接收到MESSAGE_POST_PROGRESS消息中调用`onProgressUpdate`方法。因此`onProgressUpdate`也是在主线程中调用。

小结

通过上述一步步的源码分析过程，已经掌握了AsyncTask任务的执行过程。AsyncTask中有两个线程池串行线程池sDefaultExecutor和线程池THREAD_POOL_EXECUTOR。sDefaultExecutor用于任务的排队，THREAD_POOL_EXECUTOR真正的执行任务。线程的切换使用Handler(InternalHandler)实现。

4.27 AsyncTask中的任务是串行的还是并行的？

这道题想考察什么？

是否了解AsyncTask中的任务是串行的还是并行的与真实场景使用，是否熟悉AsyncTask中的任务是串行的还是并行的

考察的知识点

AsyncTask中的任务是串行的还是并行的概念在项目中使用与基本知识

考生应该如何回答

AsyncTask中的任务在4.0以上是串行执行的，在`AsyncTask`中提交的任务默认都会通过：SerialExecutor线程池执行：

```
private static class SerialExecutor implements Executor {
    final ArrayDeque<Runnable> mTasks = new ArrayDeque<Runnable>();
    Runnable mActive;

    public synchronized void execute(final Runnable r) {
        mTasks.offer(new Runnable() {
            public void run() {
                try {
                    r.run();
                } finally {
                    scheduleNext();//当本次执行完之后，才会调用下一个执行
                }
            }
        });
        if (mActive == null) {//如果一个正在执行的都没有，那么就启动执行
            scheduleNext();
        }
    }

    protected synchronized void scheduleNext() {
        if ((mActive = mTasks.poll()) != null) {
            THREAD_POOL_EXECUTOR.execute(mActive);//将任务添加到线程池
        }
    }
}
```

通过上面的源码可以发现，每次执行完一个任务后，才会调用scheduleNext往线程池里面添加任务，所以即使线程池是并行的，但是我添加任务的时候是串行的，所以api22中的asynctask是串行的，那么线程池其实再多的线程也没用了，反正每次都只有一个任务在里面。

另外，在AsyncTask中还存在一个`executeOnExecutor(Executor exec, Params... params)`方法，能够指定线程池。所以非默认情况下，AsyncTask也能够并行执行任务。

4.28 Android中操作多线程的方式有哪些？

这道题想考察什么？

是否了解Android中操作多线程的方式有哪些与真实场景使用，是否熟悉Android中操作多线程在工作中的表现是什么？

考察的知识点

Android中操作多线程的方式有哪些的概念在项目中使用与基本知识

考生应该如何回答

常见的实现多线程的手段有五种：

第一种：Thread，Runnable

第二种：HandlerThread

第三种：AsyncTask

第四种：Executor

第五种：IntentService

Thread与Runnable

Android中创建线程最基本的两种方法，使用Thread类或Runnable接口：

> 此方式是线程最基础的用法，一般用于界面上比较简单的快捷用法，在Android中一般跟Handler一起使用，用于线程中的通信。那Android中为了方便这种通信方式，就生成了一个HandlerThread类，将Thread和Handler结合起来方便了使用。

```
/**
 * 继承Thread
 */
public class NewThread extends Thread{
    @Override
    public void run() {
        super.run();
    }
}

/**
 * 实现Runnable接口
 */
public class NewThread2 implements Runnable{
    @Override
```

```
        public void run() {

        }
    }
```

HandlerThread

HandlerThread本质上就是一个Thread，完成了对Thread与Handler的结合。主要解决的问题是，在一个已经运行的线程中去执行一些任务。
官方解释是：

```
Thread. Handy class for starting a new thread that has a looper.
The looper can then be used to create handler classes.
```

下面代码中：HandlerThread在运行中，可以通过handler进行一些任务处理。

它的原理其实就是在HandlerThread线程内部有一个Looper变量，进行loop()的死循环，然后通过MessageQueue进行一系列任务的排队和处理。
有开发者会想，这不就是普通的Thread+Looper+Handler吗，其实差不太多，HandlerThread就相当于系统帮你封装了一个带looper对象的线程，不需要你自己去手动操作Looper

那么这个HandlerThread到底有什么实际应用呢？
一般用于Android中需要新建子线程进行多个任务处理，并且需要和主线程通信。后面要说的IntentService 内部其实就是用了HandlerThread实现的。（其实我个人在实际项目中用的很少，一般用Executors.newSingleThreadExecutor()方法代替，一样的线程中管理任务队列，后面会详细说到线程池）

```
        HandlerThread mHandlerThread=new HandlerThread("");
        mHandlerThread.start();

        Handler mHandler =new Handler(mHandlerThread.getLooper()){
            @Override
            public void handleMessage(@NonNull Message msg) {
                super.handleMessage(msg);
            }
        };

        mHandler.sendEmptyMessage(0);
```

AsyncTask

AsyncTask是轻量级的异步任务类，可以在线程池中执行后台任务，然后把执行的进度和最终结果传递给主线程用于更新UI。
如果需要新建线程进行多个任务排队串行执行并且完成和主线程通信，可以使用HandlerThread。那么如果是单一任务呢，简单的任务呢？
比如我就需要请求一个接口，然后进行UI更新，那么就可以用到AsyncTask，它的优点在于**简单快捷，过程可控。**

```
new AsyncTask<Void, Void, String>() {

    @Override
```

```
    protected void onPreExecute() {
        //请求接口之前，初始化操作
        super.onPreExecute();
    }

    @Override
    protected String doInBackground(Void... parameters) {
        //请求接口
        return "";
    }

    @Override
    protected void onProgressUpdate(Void... values) {
        //在主线程显示线程任务执行的进度
        super.onProgressUpdate(values);
    }

    @Override
    protected void onPostExecute(String responseString) {
        //接收线程任务执行结果
    }
}.execute();
```

Executor

Executor即线程池，可以管理多个线程并行执行，线程池的优点就在于可以线程复用，并且合理管理所有线程。线程池具体使用与实现：**《4.22 线程池有几种实现方式，线程池的七大参数有哪些？》**

IntentService

IntentService是一个Service，自带工作线程，并且线程任务结束后自动销毁的一个类。IntentService其实封装了HandlerThread，同时又具备Service的特性。其中onHandleIntent方法即为异步执行的方法：

```
        @Override
    public void onCreate() {
              super.onCreate();
              //创建新线程并start
        HandlerThread thread = new HandlerThread("IntentService[" + mName +
"]");
        thread.start();
        mServiceLooper = thread.getLooper();
        //创建新线程对应的handler
        mServiceHandler = new ServiceHandler(mServiceLooper);
    }

    @Override
    public void onStart(@Nullable Intent intent, int startId) {
            //service启动后发送消息给handler
        Message msg = mServiceHandler.obtainMessage();
        msg.arg1 = startId;
        msg.obj = intent;
        mServiceHandler.sendMessage(msg);
    }

    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
```

```
            super(looper);
        }
        @Override
        public void handleMessage(Message msg) {
                //handler收到消息后调用onHandleIntent方法
            onHandleIntent((Intent)msg.obj);
            stopSelf(msg.arg1);
        }
    }
```

4.29 Android开发中怎样判断当前线程是否是主线程 （字节跳动）

这道题想考察什么？

是否了解怎样获取当前线程是否是主线程有哪些与真实场景使用，是否熟悉怎样获取当前线程是否是主线程在工作中的表现是什么？

考察的知识点

怎样获取当前线程是否是主线程的概念在项目中使用与基本知识

考生应该如何回答

Android开发中, 有时需要判断当前线程到底是主线程, 还是子线程, 例如: 我们在自定义View时, 想要让View重绘, 需要先判断当前线程到底是不是主线程, 然后根据判断结果来决定到底是调用 `invalidate()` 还是 `postInvalidate()` 方法。

在工作中获取当前的主线程，主要是借助Android中的Looper：

```
Looper.getMainLooper() == Looper.myLooper();
Looper.getMainLooper().getThread() == Thread.currentThread();
Looper.getMainLooper().getThread().getId() == Thread.currentThread().getId();
```

4.30 线程间如何通信？

详细讲解

享学课堂移动互联网系统课程：Android 底层FrameWork内核解析《Handler消息机制原理解析》

这道题想考察什么？

是否了解线程间如何通信与真实场景使用，是否熟悉线程间如何通信该如何操作的本质区别？

考察的知识点

Handler

考生应该如何回答

线程之间进行通信的方式需要基于具体需求分析，如果仅仅只是为了完成线程间的**同步**，使用锁如synchronized 即可；如果需要完成线程之间的协作，也可以采用wait/notify、CountDownLatch或者Cyclicbarrier等方式。而在Android中还设计了Handler机制可以完成线程间通信。

Handler是Android系统中线程间传递消息的一种机制。关于Handler的原理见享学课堂移动互联网系统课程《Handler消息机制原理解析》