Synchronized 相关问题

问题一：Synchronized 用过吗，其原理是什么？

这是一道 Java 面试中几乎百分百会问到的问题，因为没有任何写过并发程序的开发者会没听说或者没接触过 Synchronized。

Synchronized 是由 JVM 实现的一种实现互斥同步的一种方式，如果你查看被 Synchronized 修饰过的程序块编译后的字节码，会发现，被 Synchronized 修饰过的程序块，在编译前后被编译器生成了 monitorenter 和 monitorexit 两个字节码指令。

这两个指令是什么意思呢？

在虚拟机执行到 monitorenter 指令时，首先要尝试获取对象的锁：

如果这个对象没有锁定，或者当前线程已经拥有了这个对象的锁，把锁的计数器 +1；当执行 monitorexit 指令时将锁计数器 -1；当计数器为 0 时，锁就被释放了。

如果获取对象失败了，那当前线程就要阻塞等待，直到对象锁被另外一个线程释放为止。

Java 中 Synchronize 通过在对象头设置标记，达到了获取锁和释放锁的目的。

问题二：你刚才提到获取对象的锁，这个“锁”到底是什么？如何确定对象的锁？

“锁”的本质其实是 monitorenter 和 monitorexit 字节码指令的一个 Reference 类型的参数，即要锁定和解锁的对象。我们知道，使用

Synchronized 可以修饰不同的对象，因此，对应的对象锁可以这么确定。

1. 如果 Synchronized 明确指定了锁对象，比如 Synchronized（变量名）、Synchronized(this) 等，说明加解锁对象为该对象。
2. 如果没有明确指定：

   若 Synchronized 修饰的方法为非静态方法，表示此方法对应的对象为锁对象；

   若 Synchronized 修饰的方法为静态方法，则表示此方法对应的类对象为锁对象。

   注意，当一个对象被锁住时，对象里面所有用 Synchronized 修饰的方法都将产生堵塞，而对象里非 Synchronized 修饰的方法可正常被调用，不受锁影响。

问题三：什么是可重入性，为什么说 Synchronized 是可重入锁？

可重入性是锁的一个基本要求，是为了解决自己锁死自己的情况。

比如下面的伪代码，一个类中的同步方法调用另一个同步方法，假如 Synchronized 不支持重入，进入 method2 方法时当前线程获得锁，method2 方法里面执行 method1 时当前线程又要去尝试获取锁，这时如果不支持重入，它就要等释放，把自己阻塞，导致自己锁死自己。

对 Synchronized 来说，可重入性是显而易见的，刚才提到，在执行 monitorenter 指令时，如果这个对象没有锁定，或者当前线程已经拥

有了这个对象的锁（而不是已拥有了锁则不能继续获取），就把锁的计数器 +1，其实本质上就通过这种方式实现了可重入性。

问题四：JVM 对 Java 的原生锁做了哪些优化？

在 Java 6 之前，Monitor 的实现完全依赖底层操作系统的互斥锁来实现，也就是我们刚才在问题二中所阐述的获取/释放锁的逻辑。

由于 Java 层面的线程与操作系统的原生线程有映射关系，如果要将一个线程进行阻塞或唤起都需要操作系统的协助，这就需要从用户态切换到内核态来执行，这种切换代价十分昂贵，很耗处理器时间，现代 JDK 中做了大量的优化。

一种优化是使用自旋锁，即在把线程进行阻塞操作之前先让线程自旋等待一段时间，可能在等待期间其他线程已经解锁，这时就无需再让线程执行阻塞操作，避免了用户态到内核态的切换。

现代 JDK 中还提供了三种不同的 Monitor 实现，也就是三种不同的锁：

- 偏向锁（Biased Locking）
- 轻量级锁
- 重量级锁

这三种锁使得 JDK 得以优化 Synchronized 的运行，当 JVM 检测到不同的竞争状况时，会自动切换到适合的锁实现，这就是锁的升级、降级。

- 当没有竞争出现时，默认会使用偏向锁。

JVM 会利用 CAS 操作，在对象头上的 Mark Word 部分设置线程 ID，以表示这个对象偏向于当前线程，所以并不涉及真正的互斥锁，因为在很多应用场景中，大部分对象生命周期中最多会被一个线程锁定，使用偏斜锁可以降低无竞争开销。

- 如果有另一线程试图锁定某个被偏斜过的对象，JVM 就撤销偏斜锁，切换到轻量级锁实现。
- 轻量级锁依赖 CAS 操作 Mark Word 来试图获取锁，如果重试成功，就使用普通的轻量级锁；否则，进一步升级为重量级锁。

问题五：为什么说 Synchronized 是非公平锁？

非公平主要表现在获取锁的行为上，并非是按照申请锁的时间前后给等待线程分配锁的，每当锁被释放后，任何一个线程都有机会竞争到锁，这样做的目的是为了提高执行性能，缺点是可能会产生线程饥饿现象。

问题六：什么是锁消除和锁粗化？

- 锁消除：指虚拟机即时编译器在运行时，对一些代码上要求同步，但被检测到不可能存在共享数据竞争的锁进行消除。主要根据逃逸分析。

  程序员怎么会在明知道不存在数据竞争的情况下使用同步呢？很多不是程序员自己加入的。
- 锁粗化：原则上，同步块的作用范围要尽量小。但是如果一系列的连续操作都对同一个对象反复加锁和解锁，甚至加锁操作在循环体内，频繁地进行互斥同步操作也会导致不必要的性能损耗。

  锁粗化就是增大锁的作用域。

问题七：为什么说 Synchronized 是一个悲观锁？乐观锁的实现原理又是什么？什么是 CAS，它有什么特性？

Synchronized 显然是一个悲观锁，因为它的并发策略是悲观的：

不管是否会产生竞争，任何的数据操作都必须要加锁、用户态核心态转换、维护锁计数器和检查是否有被阻塞的线程需要被唤醒等操作。

随着硬件指令集的发展，我们可以使用基于冲突检测的乐观并发策略。

先进行操作，如果没有其他线程征用数据，那操作就成功了；

如果共享数据有征用，产生了冲突，那就再进行其他的补偿措施。这种乐观的并发策略的许多实现不需要线程挂起，所以被称为非阻塞同步。

乐观锁的核心算法是 CAS（Compareand Swap，比较并交换），它涉及到三个操作数：内存值、预期值、新值。当且仅当预期值和内存值相等时才将内存值修改为新值。

这样处理的逻辑是，首先检查某块内存的值是否跟之前我读取时的一样，如不一样则表示期间此内存值已经被别的线程更改过，舍弃本次操作，否则说明期间没有其他线程对此内存值操作，可以把新值设置给此块内存。

CAS 具有原子性，它的原子性由 CPU 硬件指令实现保证，即使用 JNI 调用 Native 方法调用由 C++ 编写的硬件级别指令，JDK 中提供了 Unsafe 类执行这些操作。

问题八：乐观锁一定就是好的吗？

乐观锁避免了悲观锁独占对象的现象，同时也提高了并发性能，但它也有缺点：

1. 乐观锁只能保证一个共享变量的原子操作。如果多一个或几个变量，乐观锁将变得力不从心，但互斥锁能轻易解决，不管对象数量多少及对象颗粒度大小。
2. 长时间自旋可能导致开销大。假如 CAS 长时间不成功而一直自旋，会给 CPU 带来很大的开销。
3. ABA 问题。CAS 的核心思想是通过比对内存值与预期值是否一样而判断内存值是否被改过，但这个判断逻辑不严谨，假如内存值原来是 A，后来被一条线程改为 B，最后又被改成了 A，则 CAS 认为此内存值并没有发生改变，但实际上是有被其他线程改过的，这种情况对依赖过程值的情景的运算结果影响很大。解决的思路是引入版本号，每次变量更新都把版本号加一。

可重入锁 ReentrantLock 及其他显式锁相关问题

问题一：跟 Synchronized 相比，可重入锁 ReentrantLock 其实现原理有什么不同？

其实，锁的实现原理基本是为了达到一个目的：

让所有的线程都能看到某种标记。

Synchronized 通过在对象头中设置标记实现了这一目的，是一种 JVM 原生的锁实现方式，而 ReentrantLock 以及所有的基于 Lock 接口的实现类，都是通过用一个 volitile 修饰的 int 型变量，并保证每个线程都能拥有对该 int 的可见性和原子修改，其本质是基于所谓的 AQS 框架。

问题二：那么请谈谈 AQS 框架是怎么回事儿？

AQS（AbstractQueuedSynchronizer 类）是一个用来构建锁和同步器的框架，各种 Lock 包中的锁（常用的有 ReentrantLock、ReadWriteLock），以及其他如 Semaphore、CountDownLatch，甚至是早期的 FutureTask 等，都是基于 AQS 来构建。

1. AQS 在内部定义了一个 volatile int state 变量，表示同步状态：当线程调用 lock 方法时 ，如果 state=0，说明没有任何线程占有共享资源的锁，可以获得锁并将 state=1；如果 state=1，则说明有线程目前正在使用共享变量，其他线程必须加入同步队列进行等待。
2. AQS 通过 Node 内部类构成的一个双向链表结构的同步队列，来完成线程获取锁的排队工作，当有线程获取锁失败后，就被添加到队列末尾。
   - Node 类是对要访问同步代码的线程的封装，包含了线程本身及其状态叫 waitStatus（有五种不同 取值，分别表示是否被阻塞，是否等待唤醒，是否已经被取消等），每个 Node 结点关联其 prev 结点和 next 结点，方便线程释放锁后快速唤醒下一个在等待的线程，是一个 FIFO 的过程。
   - Node 类有两个常量，SHARED 和 EXCLUSIVE，分别代表共享模式和独占模式。所谓共享模式是一个锁允许多条线程同时操作（信号量 Semaphore 就是基于 AQS 的共享模式实现的），独占模式是同一个时间段只能有一个线程对共享资源进行操作，多余的请求线程需要排队等待（如 ReentranLock）。
3. AQS 通过内部类 ConditionObject 构建等待队列（可有多个），当 Condition 调用 wait() 方法后，线程将会加入等待队列中，而当

Condition 调用 signal() 方法后，线程将从等待队列转移动同步队列中进行锁竞争。

4. AQS 和 Condition 各自维护了不同的队列，在使用 Lock 和 Condition 的时候，其实就是两个队列的互相移动。

问题三：请尽可能详尽地对比下 Synchronized 和 ReentrantLock 的异同。

ReentrantLock 是 Lock 的实现类，是一个互斥的同步锁。

从功能角度，ReentrantLock 比 Synchronized 的同步操作更精细（因为可以像普通对象一样使用），甚至实现 Synchronized 没有的高级功能，如：

- 等待可中断：当持有锁的线程长期不释放锁的时候，正在等待的线程可以选择放弃等待，对处理执行时间非常长的同步块很有用。
- 带超时的获取锁尝试：在指定的时间范围内获取锁，如果时间到了仍然无法获取则返回。
- 可以判断是否有线程在排队等待获取锁。
- 可以响应中断请求：与 Synchronized 不同，当获取到锁的线程被中断时，能够响应中断，中断异常将会被抛出，同时锁会被释放。
- 可以实现公平锁。

从锁释放角度，Synchronized 在 JVM 层面上实现的，不但可以通过一些监控工具监控 Synchronized 的锁定，而且在代码执行出现异常时，JVM 会自动释放锁定；但是使用 Lock 则不行，Lock 是通过代

码实现的，要保证锁定一定会被释放，就必须将 unLock() 放到 finally{} 中。

从性能角度，Synchronized 早期实现比较低效，对比 ReentrantLock，大多数场景性能都相差较大。

但是在 Java 6 中对其进行了非常多的改进，在竞争不激烈时，Synchronized 的性能要优于 ReetrantLock；在高竞争情况下，Synchronized 的性能会下降几十倍，但是 ReetrantLock 的性能能维持常态。

问题四：ReentrantLock 是如何实现可重入性的？

ReentrantLock 内部自定义了同步器 Sync（Sync 既实现了 AQS，又实现了 AOS，而 AOS 提供了一种互斥锁持有的方式），其实就是加锁的时候通过 CAS 算法，将线程对象放到一个双向链表中，每次获取锁的时候，看下当前维护的那个线程 ID 和当前请求的线程 ID 是否一样，一样就可重入了。

问题五：除了 ReetrantLock，你还接触过 JUC 中的哪些并发工具？

通常所说的并发包（JUC）也就是 java.util.concurrent 及其子包，集中了 Java 并发的各种基础工具类，具体主要包括几个方面：

- 提供了 CountDownLatch、CyclicBarrier、Semaphore 等，比 Synchronized 更加高级，可以实现更加丰富多线程操作的同步结构。
- 提供了 ConcurrentHashMap、有序的 ConcunrrentSkipListMap，或者通过类似快照机制实现线程安全的动态数组 CopyOnWriteArrayList 等，各种线程安全的容器。

- 提供了 ArrayBlockingQueue、SynchorousQueue 或针对特定场景的 PriorityBlockingQueue 等，各种并发队列实现。
- 强大的 Executor 框架，可以创建各种不同类型的线程池，调度任务运行等。

问题六：请谈谈 ReadWriteLock 和 StampedLock。

虽然 ReentrantLock 和 Synchronized 简单实用，但是行为上有一定局限性，要么不占，要么独占。实际应用场景中，有时候不需要大量竞争的写操作，而是以并发读取为主，为了进一步优化并发操作的粒度，Java 提供了读写锁。

读写锁基于的原理是多个读操作不需要互斥，如果读锁试图锁定时，写锁是被某个线程持有，读锁将无法获得，而只好等待对方操作结束，这样就可以自动保证不会读取到有争议的数据。

ReadWriteLock 代表了一对锁，下面是一个基于读写锁实现的数据结构，当数据量较大，并发读多、并发写少的时候，能够比纯同步版本凸显出优势：

```
public class RWSample {
 private final Map<String, String> m = new TreeMap<>();
 private final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
 private final Lock r = rwl.readLock();
 private final Lock w = rwl.writeLock();
 public String get(String key) {
     r.lock();
     System.out.println(" 读锁锁定! ");
     try {
         return m.get(key);
     } finally {
         r.unlock();
     }
 }

 public String put(String key, String entry) {
     w.lock();
     System.out.println(" 写锁锁定! ");
     try {
          return m.put(key, entry);
     } finally {
          w.unlock();
     }
  }
  // …
}
```

读写锁看起来比 Synchronized 的粒度似乎细一些，但在实际应用中，其表现也并不尽如人意，主要还是因为相对比较大的开销。

所以，JDK 在后期引入了 StampedLock，在提供类似读写锁的同时，还支持优化读模式。优化读基于假设，大多数情况下读操作并不会和写操作冲突，其逻辑是先试着修改，然后通过 validate 方法确认是否进入了写模式，如果没有进入，就成功避免了开销；如果进入，则尝试获取读锁。

```java
public class StampedSample {
 private final StampedLock sl = new StampedLock();

 void mutate() {
     long stamp = sl.writeLock();
     try {
         write();
     } finally {
         sl.unlockWrite(stamp);
     }
 }

 Data access() {
     long stamp = sl.tryOptimisticRead();
     Data data = read();
     if (!sl.validate(stamp)) {
         stamp = sl.readLock();
         try {
             data = read();
         } finally {
             sl.unlockRead(stamp);
         }
     }
     return data;
 }
 // …
}
```

问题七：如何让 Java 的线程彼此同步？你了解过哪些同步器？请分别介绍下。

JUC 中的同步器三个主要的成员：CountDownLatch、CyclicBarrier 和 Semaphore，通过它们可以方便地实现很多线程之间协作的功能。

CountDownLatch 叫倒计数，允许一个或多个线程等待某些操作完成。看几个场景：

- 跑步比赛，裁判需要等到所有的运动员（“其他线程”）都跑到终点（达到目标），才能去算排名和颁奖。
- 模拟并发，我需要启动 100 个线程去同时访问某一个地址，我希望它们能同时并发，而不是一个一个的去执行。

用法：CountDownLatch　　构造方法指明计数数量，被等待线程调用 countDown 将计数器减 1，等待线程使用 await 进行线程等待。一个简单的例子：

```java
public class TestCountDownLatch {
    private CountDownLatch countDownLatch = new CountDownLatch(4); // 构造方法指明计数数量

    public static void main(String[] args) {
        TestCountDownLatch testCountDownLatch = new TestCountDownLatch();
        testCountDownLatch.begin();
    }
    // 运动员类
    private class Runner implements Runnable {
        private int result;
        public Runner(int result) {
            this.result = result;
        }

        @Override
        public void run() {
            try {
                Thread.sleep(result * 1000); // 模拟跑了多少秒
                countDownLatch.countDown();  // 跑完了就计数器减1
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    private void begin() {
        System.out.println("赛跑开始");
        Random random = new Random(System.currentTimeMillis());
        for (int i = 0; i < 4; i++) {
            int result = random.nextInt(3) + 1; // 随机设置每个运动员跑多少秒结束
            new Thread(new Runner(result)).start();
        }
        try {
            countDownLatch.await(); // 线程等待倒数为0
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("所有人都跑完了，裁判开始算成绩");
    }
```

CyclicBarrier　叫循环栅栏，它实现让一组线程等待至某个状态之后再全部同时执行，而且当所有等待线程被释放后，CyclicBarrier　可以被重复使用。CyclicBarrier 的典型应用场景是用来等待并发线程结束。

CyclicBarrier 的主要方法是 await()，await() 每被调用一次，计数便会减少 1，并阻塞住当前线程。当计数减至 0 时，阻塞解除，所有在此 CyclicBarrier 上面阻塞的线程开始运行

在这之后，如果再次调用 await()，计数就又会变成 N-1，新一轮重新开始，这便是 Cyclic 的含义所在。CyclicBarrier.await() 带有返回值，用来表示当前线程是第几个到达这个 Barrier 的线程。

举例说明如下：

```java
public class TestCyclicBarrier {
    private CyclicBarrier cyclicBarrier = new CyclicBarrier(5);

    public static void main(String[] args) {
        new TestCyclicBarrier().begin();
    }

    public void begin() {
        for (int i = 0; i < 5; i++) {
            new Thread(new Student()).start();
        }

    }

    private class Student implements Runnable {
        @Override
        public void run() {
            try {
                Thread.sleep(2000);     // 该学生正在赶往饭店的路上
                cyclicBarrier.await(); // 到了就等着，等其他人都到了，就进饭店
            } catch (Exception e) {
                e.printStackTrace();
            }
            // TODO:大家都到了，进去吃饭吧!
        }
    }
}
```

Semaphore，Java 版本的信号量实现，用于控制同时访问的线程个数，来达到限制通用资源访问的目的，其原理是通过 acquire() 获取一个许可，如果没有就等待，而 release() 释放一个许可。

```java
public class Test {
    public static void main(String[] args) {
        Semaphore semaphore = new Semaphore(5); // 机器数目，即5个许可
        for(int i = 0; i < 8; i++)              // 工人数，8个去抢许可
            new Worker(i,semaphore).start();
    }

    static class Worker extends Thread{
        private int num;
        private Semaphore semaphore;
        public Worker(int num,Semaphore semaphore){
            this.num = num;
            this.semaphore = semaphore;
        }

        @Override
        public void run() {
            try {
                semaphore.acquire(); // 抢许可
                Thread.sleep(2000);
                semaphore.release(); // 释放许可
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

如果 Semaphore 的数值被初始化为 1，那么一个线程就可以通过 acquire 进入互斥状态，本质上和互斥锁是非常相似的。但是区别也非常明显，比如互斥锁是有持有者的，而对于 Semaphore 这种计数器结构，虽然有类似功能，但其实不存在真正意义的持有者，除非我们进行扩展包装。

问题八：CyclicBarrier 和 CountDownLatch 看起来很相似，请对比下呢？

它们的行为有一定相似度，区别主要在于：

- CountDownLatch 是不可以重置的，所以无法重用，CyclicBarrier 没有这种限制，可以重用。
- CountDownLatch 的基本操作组合是 countDown/await，调用 await 的线程阻塞等待 countDown 足够的次数，不管你是在一个线

程还是多个线程里 countDown，只要次数足够即可。 CyclicBarrier 的基本操作组合就是 await，当所有的伙伴都调用了 await，才会继续进行任务，并自动进行重置。

CountDownLatch 目的是让一个线程等待其他 N 个线程达到某个条件后，自己再去做某个事（通过 CyclicBarrier 的第二个构造方法 public CyclicBarrier(int parties, Runnable barrierAction)，在新线程里做事可以达到同样的效果）。而 CyclicBarrier 的目的是让 N 多线程互相等待直到所有的都达到某个状态，然后这 N 个线程再继续执行各自后续（通过 CountDownLatch 在某些场合也能完成类似的效果）。

Java 线程池相关问题

问题一：Java 中的线程池是如何实现的？

- 在 Java 中，所谓的线程池中的“线程”，其实是被抽象为了一个静态内部类 Worker，它基于 AQS 实现，存放在线程池的 HashSet<Worker> workers 成员变量中；
- 而需要执行的任务则存放在成员变量 workQueue（BlockingQueue<Runnable> workQueue）中。

这样，整个线程池实现的基本思想就是：从 workQueue 中不断取出需要执行的任务，放在 Workers 中进行处理。

问题二：创建线程池的几个核心构造参数？

Java 中的线程池的创建其实非常灵活，我们可以通过配置不同的参数，创建出行为不同的线程池，这几个参数包括：

- corePoolSize：线程池的核心线程数。
- maximumPoolSize：线程池允许的最大线程数。
- keepAliveTime：超过核心线程数时闲置线程的存活时间。
- workQueue：任务执行前保存任务的队列，保存由 execute 方法提交的 Runnable 任务。

问题三：线程池中的线程是怎么创建的？是一开始就随着线程池的启动创建好的吗？

显然不是的。线程池默认初始化后不启动 Worker，等待有请求时才启动。

每当我们调用 execute() 方法添加一个任务时，线程池会做如下判断：

- 如果正在运行的线程数量小于 corePoolSize，那么马上创建线程运行这个任务；
- 如果正在运行的线程数量大于或等于 corePoolSize，那么将这个任务放入队列；
- 如果这时候队列满了，而且正在运行的线程数量小于 maximumPoolSize，那么还是要创建非核心线程立刻运行这个任务；
- 如果队列满了，而且正在运行的线程数量大于或等于 maximumPoolSize，那么线程池会抛出异常 RejectExecutionException。

当一个线程完成任务时，它会从队列中取下一个任务来执行。　　当一个线程无事可做，超过一定的时间（keepAliveTime）时，线程池会判断。

如果当前运行的线程数大于　corePoolSize，那么这个线程就被停掉。所以线程池的所有任务完成后，它最终会收缩到　corePoolSize　的大小。

问题四：既然提到可以通过配置不同参数创建出不同的线程池，那么 Java 中默认实现好的线程池又有哪些呢？请比较它们的异同。

1. SingleThreadExecutor 线程池

这个线程池只有一个核心线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

- corePoolSize：1，只有一个核心线程在工作。
- maximumPoolSize：1。
- keepAliveTime：0L。
- workQueue：new LinkedBlockingQueue<Runnable>()，其缓冲队列是无界的。

2. FixedThreadPool 线程池

FixedThreadPool 是固定大小的线程池，只有核心线程。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

FixedThreadPool 多数针对一些很稳定很固定的正规并发线程，多用于服务器。

- corePoolSize：nThreads
- maximumPoolSize：nThreads
- keepAliveTime：0L
- workQueue：new LinkedBlockingQueue<Runnable>()，其缓冲队列是无界的。

3. CachedThreadPool 线程池

CachedThreadPool 是无界线程池，如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60 秒不执行任务）线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。

线程池大小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。SynchronousQueue 是一个是缓冲区为 1 的阻塞队列。

缓存型池子通常用于执行一些生存期很短的异步型任务，因此在一些面向连接的 daemon 型 SERVER 中用得不多。但对于生存期短的异步任务，它是 Executor 的首选。

- corePoolSize：0
- maximumPoolSize：Integer.MAX_VALUE
- keepAliveTime：60L
- workQueue：new SynchronousQueue<Runnable>()，一个是缓冲区为 1 的阻塞队列。

4. ScheduledThreadPool 线程池

ScheduledThreadPool：核心线程池固定，大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。创建一个周期性执行任务的线程池。如果闲置，非核心线程池会在 DEFAULT_KEEPALIVEMILLIS 时间内回收。

- corePoolSize：corePoolSize
- maximumPoolSize：Integer.MAX_VALUE
- keepAliveTime：DEFAULT_KEEPALIVE_MILLIS
- workQueue：new DelayedWorkQueue() 无界阻塞延迟队列

问题六：如何在 Java 线程池中提交线程？

线程池最常用的提交任务的方法有两种：

1. execute()：ExecutorService.execute 方法接收一个 Runable 实例，它用来执行一个任务：

```
ExecutorService.execute(Runnable runable)
```

2. submit()：ExecutorService.submit() 方法返回的是 Future 对象。可以用 isDone() 来查询 Future 是否已经完成，当任务完成时，它具有一个结果，可以调用 get() 来获取结果。也可以不用 isDone() 进行检查就直接调用 get()，在这种情况下，get() 将阻塞，直至结果准备就绪。

```
FutureTask task = ExecutorService.submit(Runnable runnable);
FutureTask<T> task = ExecutorService.submit(Runnable runnable,T Result);
FutureTask<T> task = ExecutorService.submit(Callable<T> callable);
```

Java 内存模型相关问题

问题一：什么是 Java 的内存模型，Java 中各个线程是怎么彼此看到对方的变量的？

Java 的内存模型定义了程序中各个变量的访问规则，即在虚拟机中将变量存储到内存和从内存中取出这样的底层细节。

此处的变量包括实例字段、静态字段和构成数组对象的元素，但是不包括局部变量和方法参数，因为这些是线程私有的，不会被共享，所以不存在竞争问题。

Java 中各个线程是怎么彼此看到对方的变量的呢？Java 中定义了主内存与工作内存的概念：

所有的变量都存储在主内存，每条线程还有自己的工作内存，保存了被该线程使用到的变量的主内存副本拷贝。

线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，不能直接读写主内存的变量。不同的线程之间也无法直接访问对方工作内存的变量，线程间变量值的传递需要通过主内存。

问题二：请谈谈 volatile 有什么特点，为什么它能保证变量对所有线程的可见性？

关键字 volatile 是 Java 虚拟机提供的最轻量级的同步机制。当一个变量被定义成 volatile 之后，具备两种特性：

1. 保证此变量对所有线程的可见性。当一条线程修改了这个变量的值，新值对于其他线程是可以立即得知的。而普通变量做不到这一点。

2. 禁止指令重排序优化。普通变量仅仅能保证在该方法执行过程中，得到正确结果，但是不保证程序代码的执行顺序。

Java 的内存模型定义了 8 种内存间操作：

lock 和 unlock

- 把一个变量标识为一条线程独占的状态。
- 把一个处于锁定状态的变量释放出来，释放之后的变量才能被其他线程锁定。

read 和 write

- 把一个变量值从主内存传输到线程的工作内存，以便 load。
- 把 store 操作从工作内存得到的变量的值，放入主内存的变量中。

load 和 store

- 把 read 操作从主内存得到的变量值放入工作内存的变量副本中。
- 把工作内存的变量值传送到主内存，以便 write。

use 和 assgin

- 把工作内存变量值传递给执行引擎。
- 将执行引擎值传递给工作内存变量值。

volatile 的实现基于这 8 种内存间操作，保证了一个线程对某个 volatile 变量的修改，一定会被另一个线程看见，即保证了可见性。

问题三：既然 volatile 能够保证线程间的变量可见性，是不是就意味着基于 volatile 变量的运算就是并发安全的？

显然不是的。基于 volatile 变量的运算在并发下不一定是安全的。

volatile 变量在各个线程的工作内存，不存在一致性问题（各个线程的工作内存中 volatile 变量，每次使用前都要刷新到主内存）。

但是 Java 里面的运算并非原子操作，导致 volatile 变量的运算在并发下一样是不安全的。

问题四：请对比下 volatile 对比 Synchronized 的异同。

Synchronized 既能保证可见性，又能保证原子性，而 volatile 只能保证可见性，无法保证原子性。

ThreadLocal 和 Synchonized 都用于解决多线程并发访问，防止任务在共享资源上产生冲突。但是 ThreadLocal 与 Synchronized 有本质的区别。

Synchronized 用于实现同步机制，是利用锁的机制使变量或代码块在某一时该只能被一个线程访问，是一种 “以时间换空间” 的方式。

而 ThreadLocal 为每一个线程都提供了变量的副本，使得每个线程在某一时间访问到的并不是同一个对象，根除了对变量的共享，是一种 “以空间换时间” 的方式。

问题五：请谈谈 ThreadLocal 是怎么解决并发安全的？

ThreadLocal 这是 Java 提供的一种保存线程私有信息的机制，因为其在整个线程生命周期内有效，所以可以方便地在一个线程关联的不同业务模块之间传递信息，比如事务 ID、Cookie 等上下文相关信息。

ThreadLocal 为每一个线程维护变量的副本，把共享数据的可见范围限制在同一个线程之内，其实现原理是，在 ThreadLocal 类中有一个 Map，用于存储每一个线程的变量的副本。

问题六：很多人都说要慎用 ThreadLocal，谈谈你的理解，使用 ThreadLocal 需要注意些什么？

使用 ThreadLocal 要注意 remove！

ThreadLocal 的实现是基于一个所谓的 ThreadLocalMap，在 ThreadLocalMap 中，它的 key 是一个弱引用。

通常弱引用都会和引用队列配合清理机制使用，但是 ThreadLocal 是个例外，它并没有这么做。

这意味着，废弃项目的回收依赖于显式地触发，否则就要等待线程结束，进而回收相应 ThreadLocalMap！这就是很多 OOM 的来源，所以通常都会建议，应用一定要自己负责 remove，并且不要和线程池配合，因为 worker 线程往往是不会退出的。