

## 基础知识

---

### 1. 为什么要使用并发编程

- 提升多核CPU的利用率：一般来说一台主机上的会有多个CPU核心，我们可以创建多个线程，理论上讲操作系统可以将多个线程分配给不同的CPU去执行，每个CPU执行一个线程，这样就提高了CPU的使用效率，如果使用单线程就只能有一个CPU核心被使用。
- 比如当我们在网上购物时，为了提升响应速度，需要拆分，减库存，生成订单等等这些操作，就可以进行拆分利用多线程的技术完成。面对复杂业务模型，并行程序会比串行程序更适应业务需求，而并发编程更能吻合这种业务拆分。
- 简单来说就是：
  - 充分利用多核CPU的计算能力；
  - 方便进行业务拆分，提升应用性能

### 2. 多线程应用场景

- 例如：迅雷多线程下载、数据库连接池、分批发送短信等。

### 3. 并发编程有什么缺点

- 并发编程的目的就是为了能提高程序的执行效率，提高程序运行速度，但是并发编程并不总是能提高程序运行速度的，而且并发编程可能会遇到很多问题，比如：内存泄漏、上下文切换、线程安全、死锁等问题。

### 4. 并发编程三个必要因素是什么？

- 原子性：原子，即一个不可再被分割的颗粒。原子性指的是一个或多个操作要么全部执行成功要么全部执行失败。
- 可见性：一个线程对共享变量的修改,另一个线程能够立刻看到。（synchronized,volatile）
- 有序性：程序执行的顺序按照代码的先后顺序执行。（处理器可能会对指令进行重排序）

## 5. Java 程序中怎么保证多线程的运行安全？

- 出现线程安全问题的原因一般都是三个原因：
  - 线程切换带来的原子性问题 解决办法：使用多线程之间同步synchronized或使用锁(lock)。
  - 缓存导致的可见性问题 解决办法：synchronized、volatile、LOCK，可以解决可见性问题
  - 编译优化带来的有序性问题 解决办法：Happens-Before 规则可以解决有序性问题

## 6. 并行和并发有什么区别？

- 并发：多个任务在同一个 CPU 核上，按细分的时间片轮流(交替)执行，从逻辑上来看那些任务是同时执行。
- 并行：单位时间内，多个处理器或多核处理器同时处理多个任务，是真正意义上的“同时进行”。
- 串行：有n个任务，由一个线程按顺序执行。由于任务、方法都在一个线程执行所以不存在线程不安全情况，也就不存在临界区的问题。

做一个形象的比喻：

- 并发 = 两个人用一台电脑。
- 并行 = 两个人分配了俩台电脑。
- 串行 = 两个人排队使用一台电脑。

## 7. 什么是多线程

- 多线程：多线程是指程序中包含多个执行流，即在一个程序中可以同时运行多个不同的线程来执行不同的任务。

## 8. 多线程的好处

- 可以提高 CPU 的利用率。在多线程程序中，一个线程必须等待的时候，CPU 可以运行其它的线程而不是等待，这样就大大提高了程序的效率。也就是说允许单个程序创建多个并行执行的线程来完成各自的任务。

## 9. 多线程的劣势：

- 线程也是程序，所以线程需要占用内存，线程越多占用内存也越多；
- 多线程需要协调和管理，所以需要 CPU 时间跟踪线程；
- 线程之间对共享资源的访问会相互影响，必须解决竞用共享资源的问题。

## 10. 线程和进程区别

- 什么是线程和进程？
  - 进程

一个在内存中运行的应用程序。每个正在系统上运行的程序都是一个进程

- 线程

进程中的一个执行任务（控制单元），它负责在程序里独立执行。

一个进程至少有一个线程，一个进程可以运行多个线程，多个线程可共享数据。

- 进程与线程的区别
  - 根本区别：进程是操作系统资源分配的基本单位，而线程是处理器任务调度和执行的基本单位
  - 资源开销：每个进程都有独立的代码和数据空间（程序上下文），程序之间的切换会有较大的开销；线程可以看做轻量级的进程，同一类线程共享代码和数据空间，每个线程都有自己独立的运行栈和程序计数器（PC），线程之间切换的开销小。
  - 包含关系：如果一个进程内有多条线程，则执行过程不是一条线的，而是多条线（线程）共同完成的；线程是进程的一部分，所以线程也被称为轻权进程或者轻量级进程。
  - 内存分配：同一进程的线程共享本进程的地址空间和资源，而进程与进程之间的地址空间和资源是相互独立的
  - 影响关系：一个进程崩溃后，在保护模式下不会对其他进程产生影响，但是一个线程崩溃有可能导致整个进程都死掉。所以多进程要比多线程健壮。
  - 执行过程：每个独立的进程有程序运行的入口、顺序执行序列和程序出口。但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制，两者均可并发执行

## 11. 什么是上下文切换？

- 多线程编程中一般线程的个数都大于 CPU 核心的个数，而一个 CPU 核心在任意时刻只能被一个线程使用，为了让这些线程都能得到有效执行，CPU 采取的策略是为每个线程分配时间片并轮转的形式。当一个线程的时间片用完的时候就会重新处于就绪状态让给其他线程使用，这个过程就属于一次上下文切换。
- 概括来说就是：当前任务在执行完 CPU 时间片切换到另一个任务之前会先保存自己的状态，以便下次再切换回这个任务时，可以再加载这个任务的状态。任务从保存到再加载的过程就是一次上下文切换。
- 上下文切换通常是计算密集型的。也就是说，它需要相当可观的处理器时间，在每秒几十上百次的切换中，每次切换都需要纳秒量级的时间。所以，上下文切换对系统来说意味着消耗大量的 CPU 时间，事实上，可能是操作系统中时间消耗最大的操作。
- Linux 相比与其他操作系统（包括其他类 Unix 系统）有很多的优点，其中有一项就是，其上下文切换和模式切换的时间消耗非常少。

## 12. 守护线程和用户线程有什么区别呢？

- 用户 (User) 线程：运行在前台，执行具体的任务，如程序的主线程、连接网络的子线程等都是用户线程
- 守护 (Daemon) 线程：运行在后台，为其他前台线程服务。也可以说守护线程是 JVM 中非守护线程的“佣人”。一旦所有用户线程都结束运行，守护线程会随 JVM 一起结束工作

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

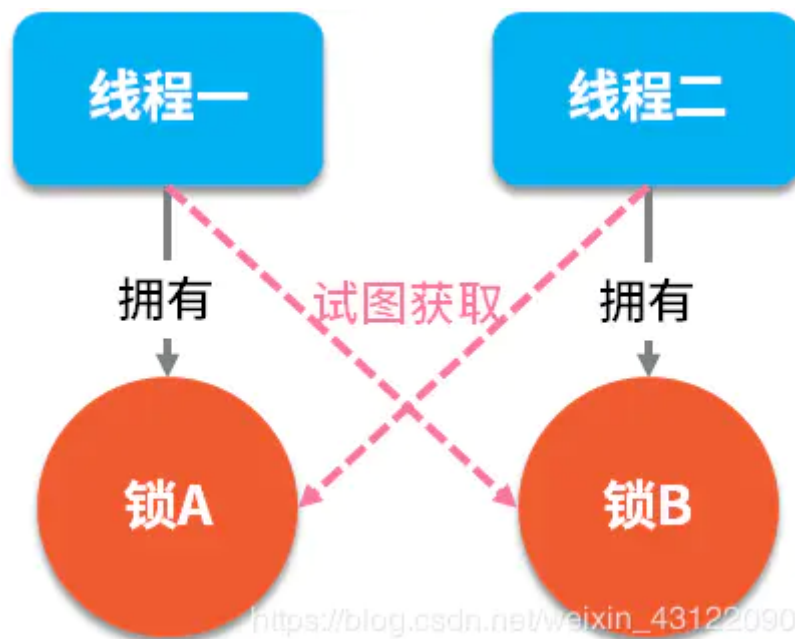
- windows上面用任务管理器看，linux下可以用 top 这个工具看。
  - 找出cpu耗用厉害的进程pid，终端执行top命令，然后按下shift+p (shift+m是找出消耗内存最高)查找出cpu利用最厉害的pid号

- 根据上面第一步拿到的pid号，`top -H -p pid`。然后按下shift+p，查找出cpu利用率最厉害的线程号，比如`top -H -p 1328`
  - 将获取到的线程号转换成16进制，去百度转换一下就行
- 使用jstack工具将进程信息打印输出，`jstack pid号 > /tmp/t.dat`，比如`jstack 31365 > /tmp/t.dat`
  - 编辑/tmp/t.dat文件，查找线程号对应的信息

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

## 14. 什么是线程死锁

- 死锁是指两个或两个以上的进程（线程）在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程（线程）称为死锁进程（线程）。
- 多个线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于线程被无限期地阻塞，因此程序不可能正常终止。
- 如下图所示，线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。



## 15. 形成死锁的四个必要条件是什么

- 互斥条件：在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，就只能等待，直至占有资源的进程用毕释放。
- 占有且等待条件：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。
- 不可抢占条件：别人已经占有了某项资源，你不能因为自己也需要该资源，就去把别人的资源抢过来。
- 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。（比如一个进程集合，A在等B，B在等C，C在等A）

## 16. 如何避免线程死锁

1. 避免一个线程同时获得多个锁
2. 避免一个线程在锁内同时占用多个资源，尽量保证每个锁只占用一个资源
3. 尝试使用定时锁，使用lock.tryLock(timeout)来替代使用内部锁机制

## 17. 创建线程的四种方式

- 继承 Thread 类;

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

- 实现 Runnable 接口;

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

- 实现 Callable 接口;

```
public class MyCallable implements Callable<Integer> {
    @Override
    public Integer call() {
        System.out.println(Thread.currentThread().getName() + " call()方法执行中...");
        return 1;
    }
}
```

- 使用匿名内部类方式

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

## 18. 说一下 runnable 和 callable 有什么区别

相同点：

- 都是接口
- 都可以编写多线程程序
- 都采用Thread.start()启动线程

主要区别：

- Runnable 接口 run 方法无返回值；Callable 接口 call 方法有返回值，是个泛型，和Future、FutureTask配合可以用来获取异步执行的结果
- Runnable 接口 run 方法只能抛出运行时异常，且无法捕获处理；Callable 接口 call 方法允许抛出异常，可以获取异常信息 注：Callable接口支持返回执行结果，需要调用FutureTask.get()得到，此方法会阻塞主进程的继续往下执行，如果不调用不会阻塞。

## 19. 线程的 run()和 start()有什么区别？

- 每个线程都是通过某个特定Thread对象所对应的方法run()来完成其操作的，run()方法称为线程体。通过调用Thread类的start()方法来启动一个线程。
- start() 方法用于启动线程，run() 方法用于执行线程的运行时代码。run() 可以重复调用，而 start() 只能调用一次。
- start()方法来启动一个线程，真正实现了多线程运行。调用start()方法无需等待run方法体代码执行完毕，可以直接继续执行其他的代码；此时线程是处于就绪状态，并没有运行。然后通过此Thread类调用方法run()来完成其运行状态，run()方法运行结束，此线程终止。然后CPU再调度其它线程。
- run()方法是在本线程里的，只是线程里的一个函数，而不是多线程的。如果直接调用run()，其实就相当于调用了普通函数而已，直接调用run()方法必须等待run()方法执行完毕才能执行下面的代码，所以执行路径还是只有一条，根本就没有线程的特征，所以在多线程执行时要使用start()方法而不是run()方法。

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

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

- new 一个 Thread，线程进入了新建状态。调用 start() 方法，会启动一个线程并使线程进入了就绪状态，当分配到时间片后就可以开始运行了。start() 会执行线程的相应准备工作，然后自动执行 run() 方法的内容，这是真正的多线程工作。
- 而直接执行 run() 方法，会把 run 方法当成一个 main 线程下的普通方法去执行，并不会在某个线程中执行它，所以这并不是多线程工作。

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

## 21. 什么是 Callable 和 Future？

- Callable 接口类似于 Runnable，从名字就可以看出来了，但是 Runnable 不会返回结果，并且无法抛出返回结果的异常，而 Callable 功能更强大一些，被线程执行后，可以返回值，这个返回值

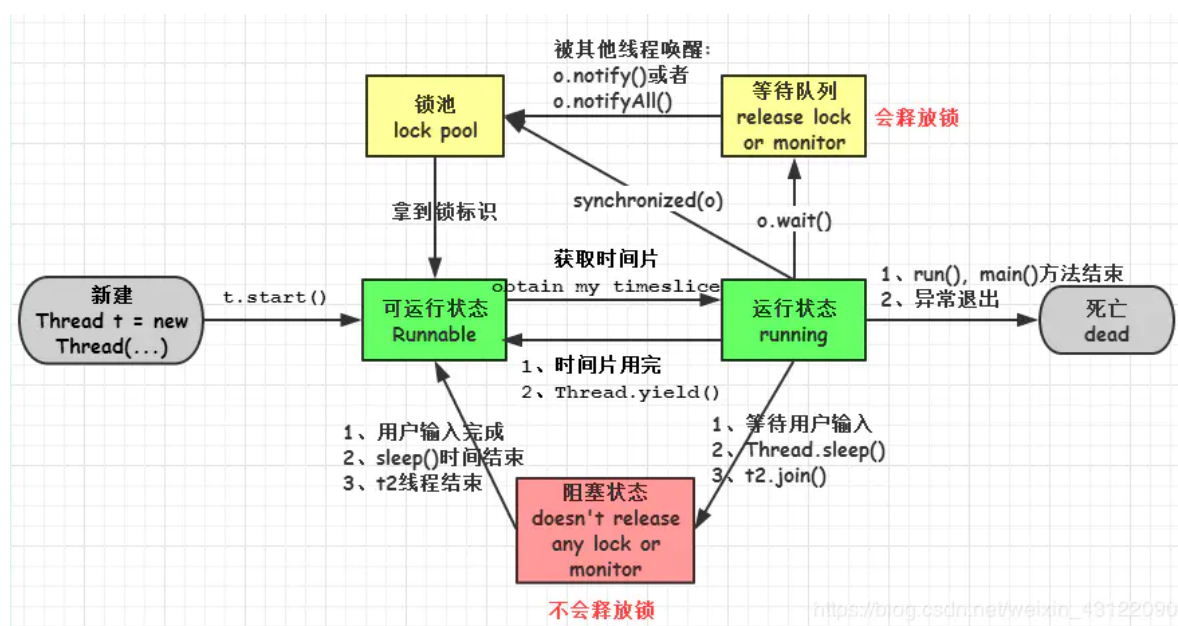
可以被 Future 拿到，也就是说，Future 可以拿到异步执行任务的返回值。

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

## 22. 什么是 FutureTask

- FutureTask 表示一个异步运算的任务。FutureTask 里面可以传入一个 Callable 的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。只有当运算完成的时候结果才能取回，如果运算尚未完成 get 方法将会阻塞。一个 FutureTask 对象可以对调用了 Callable 和 Runnable 的对象进行包装，由于 FutureTask 也是 Runnable 接口的实现类，所以 FutureTask 也可以放入线程池中。

## 23. 线程的状态



- 新建(new)：新创建了一个线程对象。
- 就绪（可运行状态）(runnable)：线程对象创建后，当调用线程对象的 start()方法，该线程处于就绪状态，等待被线程调度选中，获取cpu的使用权。
- 运行(running)：可运行状态(runnable)的线程获得了cpu时间片（timeslice），执行程序代码。  
注：就绪状态是进入到运行状态的唯一入口，也就是说，线程要想进入运行状态执行，首先必须处于就绪状态中；
- 阻塞(block)：处于运行状态中的线程由于某种原因，暂时放弃对 CPU 的使用权，停止执行，此时进入阻塞状态，直到其进入到就绪状态，才有机会再次被 CPU 调用以进入到运行状态。
  - 阻塞的情况分三种：
  - (一). 等待阻塞：运行状态中的线程执行 wait()方法，JVM会把该线程放入等待队列(waitting queue)中，使本线程进入到等待阻塞状态；
  - (二). 同步阻塞：线程在获取 synchronized 同步锁失败(因为锁被其它线程所占用)，，则JVM会把该线程放入锁池(lock pool)中，线程会进入同步阻塞状态；
  - (三). 其他阻塞: 通过调用线程的 sleep()或 join()或发出了 I/O 请求时，线程会进入到阻塞状态。当 sleep()状态超时、join()等待线程终止或者超时、或者 I/O 处理完毕时，线程重新转入就绪状态。
- 死亡(dead)(结束)：线程run()、main()方法执行结束，或者因异常退出了run()方法，则该线程结束生命周期。死亡的线程不可再次复生。



## 24. Java 中用到的线程调度算法是什么？

- 计算机通常只有一个 CPU，在任意时刻只能执行一条机器指令，每个线程只有获得 CPU 的使用权才能执行指令。所谓多线程的并发运行，其实是指从宏观上看，各个线程轮流获得 CPU 的使用权，分别执行各自的任務。在运行池中，会有多个处于就绪状态的线程在等待 CPU，JAVA 虚拟机的一项任务就是负责线程的调度，线程调度是指按照特定机制为多个线程分配 CPU 的使用权。  
(Java是由JVM中的线程计数器来实现线程调度)
- 有两种调度模型：分时调度模型和抢占式调度模型。
  - 分时调度模型是指让所有的线程轮流获得 cpu 的使用权，并且平均分配每个线程占用的 CPU 的时间片这个也比较好理解。
  - Java虚拟机采用抢占式调度模型，是指优先让可运行池中优先级高的线程占用CPU，如果可运行池中的线程优先级相同，那么就随机选择一个线程，使其占用CPU。处于运行状态的线程会一直运行，直至它不得不放弃 CPU。

## 25. 线程的调度策略

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

- (1) 线程体中调用了 yield 方法让出了对 cpu 的占用权利
- (2) 线程体中调用了 sleep 方法使线程进入睡眠状态
- (3) 线程由于 IO 操作受到阻塞
- (4) 另外一个更高优先级线程出现
- (5) 在支持时间片的系统中，该线程的时间片用完

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

- 线程调度器是一个操作系统服务，它负责为 Runnable 状态的线程分配 CPU 时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。
- 时间分片是指将可用的 CPU 时间分配给可用的 Runnable 线程的过程。分配 CPU 时间可以基于线程优先级或者线程等待的时间。
- 线程调度并不受到 Java 虚拟机控制，所以由应用程序来控制它是更好的选择（也就是说不要让你的程序依赖于线程的优先级）。

## 27. 请说出与线程同步以及线程调度相关的方法。

- (1) wait(): 使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；
- (2) sleep(): 使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理 InterruptedException 异常；
- (3) notify(): 唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由 JVM 确定唤醒哪个线程，而且与优先级无关；
- (4) notifyAll(): 唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

## 28. sleep() 和 wait() 有什么区别？

两者都可以暂停线程的执行



- 类的不同：sleep() 是 Thread线程类的静态方法，wait() 是 Object类的方法。
- 是否释放锁：sleep() 不释放锁；wait() 释放锁。
- 用途不同：Wait 通常被用于线程间交互/通信，sleep 通常被用于暂停执行。
- 用法不同：wait() 方法被调用后，线程不会自动苏醒，需要别的线程调用同一个对象上的 notify() 或者 notifyAll() 方法。sleep() 方法执行完成后，线程会自动苏醒。或者可以使用wait(long timeout)超时后线程会自动苏醒。

## 29. 你是如何调用 wait() 方法的？使用 if 块还是循环？为什么？

- 处于等待状态的线程可能会收到错误警报和伪唤醒，如果不在循环中检查等待条件，程序就会在没有满足结束条件的情况下退出。
- wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没有满足，所以在处理前，循环检测条件是否满足会更好。下面是一段标准的使用 wait 和 notify 方法的代码：

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

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

- 因为Java所有类的都继承了Object，Java想让任何对象都可以作为锁，并且 wait(), notify()等方法用于等待对象的锁或者唤醒线程，在 Java 的线程中并没有可供任何对象使用的锁，所以任意对象调用方法一定定义在Object类中。
- 有的人会说，既然是线程放弃对象锁，那也可以把wait()定义在Thread类里面啊，新定义的线程继承于Thread类，也不需要重新定义wait()方法的实现。然而，这样做有一个非常大的问题，一个线程完全可以持有很多锁，你一个线程放弃锁的时候，到底要放弃哪个锁？当然了，这种设计并不是不能实现，只是管理起来更加复杂。

## 31. 为什么 wait(), notify()和 notifyAll()必须在同步方法或者同步块中被调用？

- 当一个线程需要调用对象的 wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的 notify()方法。同样的，当一个线程需要调用对象的 notify()方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。

## 32. Thread 类中的 yield 方法有什么作用？

- 使当前线程从执行状态（运行状态）变为可执行态（就绪状态）。
- 当前线程到了就绪状态，那么接下来哪个线程会从就绪状态变成执行状态呢？可能是当前线程，也可能是其他线程，看系统的分配了。

### 33. 为什么 Thread 类的 sleep()和 yield ()方法是静态的？

- Thread 类的 sleep()和 yield()方法将在当前正在执行的线程上运行。所以在其他处于等待状态的线程上调用这些方法是没有意义的。这就是为什么这些方法是静态的。它们可以在当前正在执行的线程中工作，并避免程序员错误的认为可以在其他非运行线程调用这些方法。

### 34. 线程的 sleep()方法和 yield()方法有什么区别？

- (1) sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；
- (2) 线程执行 sleep()方法后转入阻塞 (blocked) 状态，而执行 yield()方法后转入就绪 (ready) 状态；
- (3) sleep()方法声明抛出 InterruptedException，而 yield()方法没有声明任何异常；
- (4) sleep()方法比 yield()方法（跟操作系统 CPU 调度相关）具有更好的可移植性，通常不建议使用yield()方法来控制并发线程的执行。

### 35. 如何停止一个正在运行的线程？

- 在java中有以下3种方法可以终止正在运行的线程：
  - 使用退出标志，使线程正常退出，也就是当run方法完成后线程终止。
  - 使用stop方法强行终止，但是不推荐这个方法，因为stop和suspend及resume一样都是过期作废的方法。
  - 使用interrupt方法中断线程。

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

- interrupt：用于中断线程。调用该方法的线程的状态为将被置为“中断”状态。

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

- interrupted：是静态方法，查看当前中断信号是true还是false并且清除中断信号。如果一个线程被中断了，第一次调用 interrupted 则返回 true，第二次和后面的就返回 false 了。
- isInterrupted：是可以返回当前中断信号是true还是false，与interrupt最大的差别

### 37. 什么是阻塞式方法？

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

### 38. Java 中你怎样唤醒一个阻塞的线程？

- 首先，wait()、notify() 方法是针对对象的，调用任意对象的 wait()方法都将导致线程阻塞，阻塞的同时也将释放该对象的锁，相应地，调用任意对象的 notify()方法则将随机解除该对象阻塞的线

程，但它需要重新获取该对象的锁，直到获取成功才能往下执行；

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

### 39. notify() 和 notifyAll() 有什么区别？

- 如果线程调用了对象的 wait()方法，那么线程便会处于该对象的等待池中，等待池中的线程不会去竞争该对象的锁。
- notifyAll() 会唤醒所有的线程，notify() 只会唤醒一个线程。
- notifyAll() 调用后，会将全部线程由等待池移到锁池，然后参与锁的竞争，竞争成功则继续执行，如果不成功则留在锁池等待锁被释放后再次参与竞争。而 notify()只会唤醒一个线程，具体唤醒哪一个线程由虚拟机控制。

### 40. 如何在两个线程间共享数据？

- 在两个线程间共享变量即可实现共享。

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

### 41. Java 如何实现多线程之间的通讯和协作？

- 可以通过中断 和 共享变量的方式实现线程间的通讯和协作
- 比如说最经典的生产者-消费者模型：当队列满时，生产者需要等待队列有空间才能继续往里面放入商品，而在等待的期间内，生产者必须释放对临界资源（即队列）的占用权。因为生产者如果不释放对临界资源的占用权，那么消费者就无法消费队列中的商品，就不会让队列有空间，那么生产者就会一直无限等待下去。因此，一般情况下，当队列满时，会让生产者交出对临界资源的占用权，并进入挂起状态。然后等待消费者消费了商品，然后消费者通知生产者队列有空间了。同样地，当队列空时，消费者也必须等待，等待生产者通知它队列中有商品了。这种互相通信的过程就是线程间的协作。
- Java中线程通信协作的最常见方式：
  - 一.synchronoized加锁的线程的Object类的wait()/notify()/notifyAll()
  - 二.ReentrantLock类加锁的线程的Condition类的await()/signal()/signalAll()
- 线程间直接的数据交换：
  - 三.通过管道进行线程间通信：字节流、字符流

### 42. 同步方法和同步块，哪个是更好的选择？

- 同步块是更好的选择，因为它不会锁住整个对象（当然你也可以让它锁住整个对象）。同步方法会锁住整个对象，哪怕这个类中有多个不相关联的同步块，这通常会导致他们停止执行并需要等待获得这个对象上的锁。
- 同步块更要符合开放调用的原则，只在需要锁住的代码块锁住相应的对象，这样从侧面来说也可以避免死锁。

请知道一条原则：同步的范围越小越好。

### 43. 什么是线程同步和线程互斥，有哪几种实现方式？

- 当一个线程对共享的数据进行操作时，应使之成为一个“原子操作”，即在没有完成相关操作之前，不允许其他线程打断它，否则，就会破坏数据的完整性，必然会得到错误的处理结果，这就是线程的同步。
- 在多线程应用中，考虑不同线程之间的数据同步和防止死锁。当两个或多个线程之间同时等待对方释放资源的时候就会形成线程之间的死锁。为了防止死锁的发生，需要通过同步来实现线程安全。
- 线程互斥是指对于共享的进程系统资源，在各单个线程访问时的排它性。当有若干个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步。
- 线程间的同步方法大体可分为两类：用户模式和内核模式。顾名思义，内核模式就是指利用系统内核对象的单一性来进行同步，使用时需要切换内核态与用户态，而用户模式就是不需要切换到内核态，只在用户态完成操作。
- 用户模式下的方法有：原子操作（例如一个单一的全局变量），临界区。内核模式下的方法有：事件，信号量，互斥量。
- 实现线程同步的方法
  - 同步代码方法：synchronized 关键字修饰的方法
  - 同步代码块：synchronized 关键字修饰的代码块
  - 使用特殊变量域volatile实现线程同步：volatile关键字为域变量的访问提供了一种免锁机制
  - 使用重入锁实现线程同步：reentrantlock类是可冲入、互斥、实现了lock接口的锁他与synchronized方法具有相同的基本行为和语义

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

- 在 java 虚拟机中，监视器和锁在Java虚拟机中是一块使用的。监视器监视一块同步代码块，确保一次只有一个线程执行同步代码块。每一个监视器都和一个对象引用相关联。线程在获取锁之前不允许执行同步代码。
- 一旦方法或者代码块被 synchronized 修饰，那么这个部分就放入了监视器的监视区域，确保一次只能有一个线程执行该部分的代码，线程在获取锁之前不允许执行该部分的代码
- 另外 java 还提供了显式监视器( Lock )和隐式监视器( synchronized )两种锁方案

### 45. 如果你提交任务时，线程池队列已满，这时会发生什么

- 有两种可能：

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

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

### 46. 什么叫线程安全？servlet 是线程安全吗？

- 线程安全是编程中的术语，指某个方法在多线程环境中被调用时，能够正确地处理多个线程之间的共享变量，使程序功能正确完成。
- Servlet 不是线程安全的，servlet 是单实例多线程的，当多个线程同时访问同一个方法，是不能保证共享变量的线程安全性的。
- Struts2 的 action 是多实例多线程的，是线程安全的，每个请求过来都会 new 一个新的 action 分配给这个请求，请求完成后销毁。
- SpringMVC 的 Controller 是线程安全的吗？不是的，和 Servlet 类似的处理流程。
- Struts2 好处是不用考虑线程安全问题；Servlet 和 SpringMVC 需要考虑线程安全问题，但是性能可以提升不用处理太多的 gc，可以使用 ThreadLocal 来处理多线程的问题。

## 47. 在 Java 程序中怎么保证多线程的运行安全？

- 方法一：使用安全类，比如 java.util.concurrent 下的类，使用原子类 AtomicInteger
- 方法二：使用自动锁 synchronized。
- 方法三：使用手动锁 Lock。
- 手动锁 Java 示例代码如下：

```
Lock lock = new ReentrantLock();
lock.lock();
try {
    System.out.println("获得锁");
} catch (Exception e) {
    // TODO: handle exception
} finally {
    System.out.println("释放锁");
    lock.unlock();
}
```

## 48. 你对线程优先级的理解是什么？

- 每一个线程都是有优先级的，一般来说，高优先级的线程在运行时会具有优先权，但这依赖于线程调度的实现，这个实现是和操作系统相关的(OS dependent)。我们可以定义线程的优先级，但是这并不能保证高优先级的线程会在低优先级的线程前执行。线程优先级是一个 int 变量(从 1-10)，1 代表最低优先级，10 代表最高优先级。
- Java 的线程优先级调度会委托给操作系统去处理，所以与具体的操作系统优先级有关，如非特别需要，一般无需设置线程优先级。
- 当然，如果你真的想设置优先级可以通过 setPriority() 方法设置，但是设置了不一定会该变，这个是不准确的

## 49. 线程类的构造方法、静态块是被哪个线程调用的

- 这是一个非常刁钻和狡猾的问题。请记住：线程类的构造方法、静态块是被 new 这个线程类所在的线程所调用的，而 run 方法里面的代码才是被线程自身所调用的。
- 如果说上面的说法让你感到困惑，那么我举个例子，假设 Thread2 中 new 了 Thread1，main 函数中 new 了 Thread2，那么：

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

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

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

- Dump文件是进程的内存镜像。可以把程序的执行状态通过调试器保存到dump文件中。
- 在 Linux 下，你可以通过命令 kill -3 PID（Java 进程的进程 ID）来获取 Java应用的 dump 文件。
- 在 Windows 下，你可以按下 Ctrl + Break 来获取。这样 JVM 就会将线程的 dump 文件打印到标准输出或错误文件中，它可能打印在控制台或者日志文件中，具体位置依赖应用的配置。

## 51. 一个线程运行时发生异常会怎样？

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

## 52. Java 线程数过多会造成什么异常？

- 线程的生命周期开销非常高
- 消耗过多的 CPU

资源如果可运行的线程数量多于可用处理器的数量，那么有线程将会被闲置。大量空闲的线程会占用许多内存，给垃圾回收器带来压力，而且大量的线程在竞争 CPU资源时还将产生其他性能的开销。

- 降低稳定性JVM

在可创建线程的数量上存在一个限制，这个限制值将随着平台的不同而不同，并且承受着多个因素制约，包括 JVM 的启动参数、Thread 构造函数中请求栈的大小，以及底层操作系统对线程的限制等。如果破坏了这些限制，那么可能抛出OutOfMemoryError 异常。

## 53. 多线程的常用方法



| 方法名             | 描述           |
|-----------------|--------------|
| sleep()         | 强迫一个线程睡眠N毫秒  |
| isAlive()       | 判断一个线程是否存活。  |
| join()          | 等待线程终止。      |
| activeCount()   | 程序中活跃的线程数。   |
| enumerate()     | 枚举程序中的线程。    |
| currentThread() | 得到当前线程。      |
| isDaemon()      | 一个线程是否为守护线程。 |
| setDaemon()     | 设置一个线程为守护线程。 |
| setName()       | 为线程设置一个名称。   |
| wait()          | 强迫一个线程等待。    |
| notify()        | 通知一个线程继续运行。  |
| setPriority()   | 设置一个线程的优先级。  |

## 并发理论

### 1. Java中垃圾回收有什么目的？什么时候进行垃圾回收？

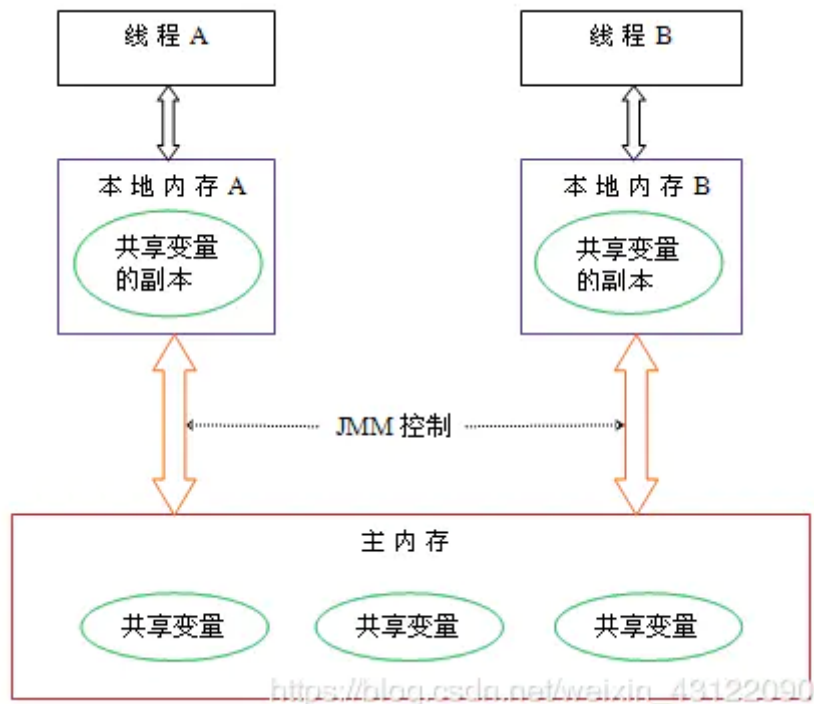
- 垃圾回收是在内存中存在没有引用的对象或超过作用域的对象时进行的。
- 垃圾回收的目的是识别并且丢弃应用不再使用的对象来释放和重用资源。

### 2. 线程之间如何通信及线程之间如何同步

- 在并发编程中，我们需要处理两个关键问题：线程之间如何通信及线程之间如何同步。通信是指线程之间以如何来交换信息。一般线程之间的通信机制有两种：共享内存和消息传递。
- Java的并发采用的是共享内存模型，Java线程之间的通信总是隐式进行，整个通信过程对程序员完全透明。如果编写多线程程序的Java程序员不理解隐式进行的线程之间通信的工作机制，很可能会遇到各种奇怪的内存可见性问题。

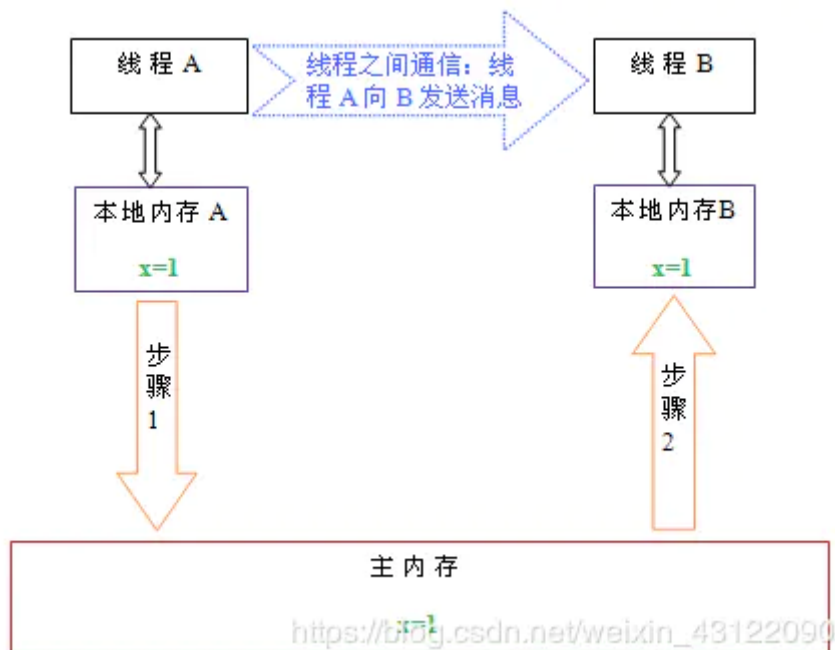
### 3. Java内存模型

- 共享内存模型指的就是Java内存模型(简称JMM)，JMM决定一个线程对共享变量的写入时,能对另一个线程可见。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他硬件和编译器优化。



- 从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：
  1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。
  2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

下面通过示意图来说明线程之间的通信



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

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

- 不会，在下一个垃圾回调周期中，这个对象将是可回收的。
- 也就是说并不会立即被垃圾收集器立刻回收，而是在下一次垃圾回收时才会释放其占用的内存。

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

- 1. 垃圾回收器 (garbage collector) 决定回收某对象时，就会运行该对象的finalize()方法；finalize是Object类的一个方法，该方法在Object类中的声明protected void finalize() throws Throwable { } 在垃圾回收器执行时会调用被回收对象的finalize()方法，可以覆盖此方法来实现对其资源的回收。注意：一旦垃圾回收器准备释放对象占用的内存，将首先调用该对象的finalize()方法，并且下一次垃圾回收动作发生时，才真正回收对象占用的内存空间
- 1. GC本来就是内存回收了，应用还需要在finalization做什么呢？答案是大部分时候，什么都不用做(也就是不需要重载)。只有在某些很特殊的情况下，比如你调用了一些native的方法(一般是C写的)，可以要在finalization里去调用C的释放函数。
  - Finalization主要用来释放被对象占用的资源(不是指内存，而是指其他资源，比如文件(File Handle)、端口(ports)、数据库连接(DB Connection)等)。然而，它不能真正有效地工作。

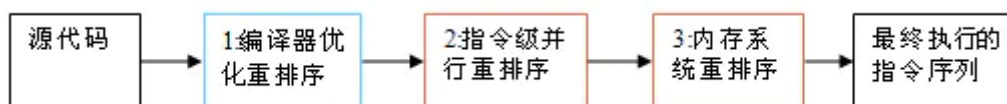
## 6. 什么是重排序

- 程序执行的顺序按照代码的先后顺序执行。
- 一般来说处理器为了提高程序运行效率，可能会对输入代码进行优化，进行重新排序(重排序)，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。

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

- 则因为重排序，他还可能执行顺序为(这里标注的是语句的执行顺序) 2-1-3-4, 1-3-2-4 但绝不可能 2-1-4-3，因为这打破了依赖关系。
- 显然重排序对单线程运行是不会有任问题，但是多线程就不一定了，所以我们在多线程编程时就得考虑这个问题了。

## 7. 重排序实际执行的指令步骤



- 编译器优化的重排序。编译器在不改变单线程程序语义的前提下，可以重新安排语句的执行顺序。
  - 指令级并行的重排序。现代处理器采用了指令级并行技术 (ILP) 来将多条指令重叠执行。如果不存在数据依赖性，处理器可以改变语句对应机器指令的执行顺序。
  - 内存系统的重排序。由于处理器使用缓存和读/写缓冲区，这使得加载和存储操作看上去可能是在乱序执行。
- 这些重排序对于单线程没问题，但是多线程都可能会导致多线程程序出现内存可见性问题。

## 8. 重排序遵守的规则

- as-if-serial:

1. 不管怎么排序，结果不能改变
2. 不存在数据依赖的可以被编译器和处理器重排序
3. 一个操作依赖两个操作，这两个操作如果不存在依赖可以重排序
4. 单线程根据此规则不会有问題，但是重排序后多线程会有问題

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

- as-if-serial语义保证单线程内程序的执行结果不被改变，happens-before关系保证正确同步的多线程程序的执行结果不被改变。
- as-if-serial语义给编写单线程程序的程序员创造了一个幻境：单线程程序是按程序的顺序来执行的。happens-before关系给编写正确同步的多线程程序的程序员创造了一个幻境：正确同步的多线程程序是按happens-before指定的顺序来执行的。
- as-if-serial语义和happens-before这么做的目的，都是为了在不改变程序执行结果的前提下，尽可能地提高程序执行的并行度。

## 10. 并发关键字 synchronized ?

- 在 Java 中，synchronized 关键字是用来控制线程同步的，就是在多线程的环境下，控制 synchronized 代码段不被多个线程同时执行。synchronized 可以修饰类、方法、变量。
- 另外，在 Java 早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后 Java 官方对从 JVM 层面对 synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

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

**synchronized关键字最主要的三种使用方式：**

- 修饰实例方法: 作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁
- 修饰静态方法: 也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份）。所以如果一个线程A调用一个实例对象的非静态 synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。
- 修饰代码块: 指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。

总结：synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是给 Class 类上锁。synchronized 关键字加到实例方法上是给对象实例上锁。尽量不要使用 synchronized(String

a) 因为JVM中，字符串常量池具有缓存功能！

## 12. 单例模式了解吗？给我解释一下双重检验锁方式实现单例模式！”

**双重校验锁实现对象单例（线程安全）**

## 说明:

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

```
public class Singleton {
    private volatile static Singleton uniqueInstance;
    private Singleton() {}

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

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

- `uniqueInstance` 采用 `volatile` 关键字修饰也是很有必要的，`uniqueInstance = new Singleton();` 这段代码其实是分为三步执行：
  1. 为 `uniqueInstance` 分配内存空间
  2. 初始化 `uniqueInstance`
  3. 将 `uniqueInstance` 指向分配的内存地址

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

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

## 13. 说一下 synchronized 底层实现原理？

- Synchronized的语义底层是通过一个monitor（监视器锁）的对象来完成，
- 每个对象有一个监视器锁(monitor)。每个Synchronized修饰过的代码当它的monitor被占用时就会处于锁定状态并且尝试获取monitor的所有权，过程：
  - 1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
  - 2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1。
  - 3、如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

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

## 14. synchronized可重入的原理

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

## 15. 什么是自旋

- 很多 synchronized 里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然 synchronized 里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在 synchronized 的边界做忙循环，这就是自旋。如果做了多次循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。
- 忙循环：就是程序员用循环让一个线程等待，不像传统方法wait(), sleep() 或 yield() 它们都放弃了CPU控制，而忙循环不会放弃CPU，它就是在运行一个空循环。这么做的目的是为了保留CPU缓存，在多核系统中，一个等待线程醒来的时候可能会在另一个内核运行，这样会重建缓存。为了避免重建缓存和减少等待重建的时间就可以使用它了。

## 16. 多线程中 synchronized 锁升级的原理是什么？

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

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

- 偏向锁，顾名思义，它会偏向于第一个访问锁的线程，如果在运行过程中，同步锁只有一个线程访问，不存在多线程争用的情况，则线程是不需要触发同步的，减少加锁 / 解锁的一些CAS操作（比如等待队列的一些CAS操作），这种情况下，就会给线程加一个偏向锁。如果在运行过程中，遇到了其他线程抢占锁，则持有偏向锁的线程会被挂起，JVM会消除它身上的偏向锁，将锁恢复到标准的轻量级锁。
- 轻量级锁是由偏向所升级来的，偏向锁运行在一个线程进入同步块的情况下，当第二个线程加入锁争用的时候，轻量级锁就会升级为重量级锁；
- 重量级锁是synchronized，是Java虚拟机中最为基础的锁实现。在这种状态下，Java虚拟机会阻塞加锁失败的线程，并且在目标锁被释放的时候，唤醒这些线程。

## 17. 线程 B 怎么知道线程 A 修改了变量

- (1) volatile 修饰变量
- (2) synchronized 修饰修改变量的方法
- (3) wait/notify
- (4) while 轮询



## 18. 当一个线程进入一个对象的 synchronized 方法 A 之后，其它线程是否可进入此对象的 synchronized 方法 B？

- 不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的 synchronized 修饰符要求执行方法时要获得对象的锁，如果已经进入A 方法说明对象锁已经被取走，那么试图进入 B 方法的线程就只能在等锁池（注意不是等待池哦）中等待对象的锁。

## 19. synchronized、volatile、CAS 比较

- (1) synchronized 是悲观锁，属于抢占式，会引起其他线程阻塞。
- (2) volatile 提供多线程共享变量可见性和禁止指令重排序优化。
- (3) CAS 是基于冲突检测的乐观锁（非阻塞）

## 20. synchronized 和 Lock 有什么区别？

- 首先synchronized是Java内置关键字，在JVM层面，Lock是个Java类；
- synchronized 可以给类、方法、代码块加锁；而 lock 只能给代码块加锁。
- synchronized 不需要手动获取锁和释放锁，使用简单，发生异常会自动释放锁，不会造成死锁；而 lock 需要自己加锁和释放锁，如果使用不当没有 unlock()去释放锁就会造成死锁。
- 通过 Lock 可以知道有没有成功获取锁，而 synchronized 却无法办到。

## 21. synchronized 和 ReentrantLock 区别是什么？

- synchronized 是和 if、else、for、while 一样的关键字，ReentrantLock 是类，这是二者的本质区别。既然 ReentrantLock 是类，那么它就提供了比synchronized 更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量
- synchronized 早期的实现比较低效，对比 ReentrantLock，大多数场景性能都相差较大，但是在 Java 6 中对 synchronized 进行了非常多的改进。
- 相同点：两者都是可重入锁

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

- 主要区别如下：
  - ReentrantLock 使用起来比较灵活，但是必须有释放锁的配合动作；
  - ReentrantLock 必须手动获取与释放锁，而 synchronized 不需要手动释放和开启锁；
  - ReentrantLock 只适用于代码块锁，而 synchronized 可以修饰类、方法、变量等。
  - 二者的锁机制其实也是不一样的。ReentrantLock 底层调用的是 Unsafe 的 park 方法加锁，synchronized 操作的应该是对象头中 mark word
- Java中每一个对象都可以作为锁，这是synchronized实现同步的基础：
  - 普通同步方法，锁是当前实例对象
  - 静态同步方法，锁是当前类的class对象
  - 同步方法块，锁是括号里面的对象

## 22. volatile 关键字的作用

- 对于可见性，Java 提供了 `volatile` 关键字来保证可见性和禁止指令重排。`volatile` 提供 `happens-before` 的保证，确保一个线程的修改能对其他线程是可见的。当一个共享变量被 `volatile` 修饰时，它会保证修改的值会立即被更新到主内存中，当有其他线程需要读取时，它会去内存中读取新值。
- 从实践角度而言，`volatile` 的一个重要作用就是和 CAS 结合，保证了原子性，详细的可以参见 `java.util.concurrent.atomic` 包下的类，比如 `AtomicInteger`。
- `volatile` 常用于多线程环境下的单次操作(单次读或者单次写)。

## 23. Java 中能创建 `volatile` 数组吗？

- 能，Java 中可以创建 `volatile` 类型数组，不过只是一个指向数组的引用，而不是整个数组。意思是，如果改变引用指向的数组，将会受到 `volatile` 的保护，但是如果多个线程同时改变数组的元素，`volatile` 标示符就不能起到之前的保护作用了。

## 24. `volatile` 变量和 `atomic` 变量有什么不同？

- `volatile` 变量可以确保先行关系，即写操作会发生在后续的读操作之前，但它并不能保证原子性。例如用 `volatile` 修饰 `count` 变量，那么 `count++` 操作就不是原子性的。
- 而 `AtomicInteger` 类提供的 `atomic` 方法可以让这种操作具有原子性如 `getAndIncrement()` 方法会原子性的进行增量操作把当前值加一，其它数据类型和引用变量也可以进行相似操作。

## 25. `volatile` 能使得一个非原子操作变成原子操作吗？

- 关键字 `volatile` 的主要作用是使变量在多个线程间可见，但无法保证原子性，对于多个线程访问同一个实例变量需要加锁进行同步。
- 虽然 `volatile` 只能保证可见性不能保证原子性，但用 `volatile` 修饰 `long` 和 `double` 可以保证其操作原子性。

所以从 Oracle Java Spec 里面可以看到：

- 对于 64 位的 `long` 和 `double`，如果没有被 `volatile` 修饰，那么对其操作可以不是原子的。在操作的时候，可以分成两步，每次对 32 位操作。
- 如果使用 `volatile` 修饰 `long` 和 `double`，那么其读写都是原子操作
- 对于 64 位的引用地址的读写，都是原子操作
- 在实现 JVM 时，可以自由选择是否把读写 `long` 和 `double` 作为原子操作
- 推荐 JVM 实现为原子操作

## 26. `synchronized` 和 `volatile` 的区别是什么？

- `synchronized` 表示只有一个线程可以获取作用对象的锁，执行代码，阻塞其他线程。
- `volatile` 表示变量在 CPU 的寄存器中是不确定的，必须从主存中读取。保证多线程环境下变量的可见性；禁止指令重排序。

### 区别

- `volatile` 是变量修饰符；`synchronized` 可以修饰类、方法、变量。
- `volatile` 仅能实现变量的修改可见性，不能保证原子性；而 `synchronized` 则可以保证变量的修改可见性和原子性。
- `volatile` 不会造成线程的阻塞；`synchronized` 可能会造成线程的阻塞。
- `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。

- volatile关键字是线程同步的轻量级实现，所以volatile性能肯定比synchronized关键字要好。但是volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块。synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，实际开发中使用synchronized关键字的场景还是更多一些。

## 27. final不可变对象，它对写并发应用有什么帮助？

- 不可变对象(Immutable Objects)即对象一旦被创建它的状态（对象的数据，也即对象属性值）就不能改变，反之即为可变对象(Mutable Objects)。
- 不可变对象的类即为不可变类(Immutable Class)。Java 平台类库中包含许多不可变类，如String、基本类型的包装类、BigInteger 和 BigDecimal 等。
- 只有满足如下状态，一个对象才是不可变的；
  - 它的状态不能在创建后再被修改；
  - 所有域都是 final 类型；并且，它被正确创建（创建期间没有发生 this 引用的逸出）。

不可变对象保证了对象的内存可见性，对不可变对象的读取不需要进行额外的同步手段，提升了代码执行效率。

## 28. Lock 接口和synchronized 对比同步它有什么优势？

- Lock 接口比同步方法和同步块提供了更具扩展性的锁操作。他们允许更灵活的结构，可以具有完全不同的性质，并且可以支持多个相关类的条件对象。
- 它的优势有：
  - (1) 可以使锁更公平
  - (2) 可以使线程在等待锁的时候响应中断
  - (3) 可以让线程尝试获取锁，并在无法获取锁的时候立即返回或者等待一段时间
  - (4) 可以在不同的范围，以不同的顺序获取和释放锁
- 整体上来说 Lock 是 synchronized 的扩展版，Lock 提供了无条件的、可轮询的(tryLock 方法)、定时的(tryLock 带参方法)、可中断的(lockInterruptibly)、可多条件队列的(newCondition 方法)锁操作。另外 Lock 的实现类基本都支持非公平锁(默认)和公平锁，synchronized 只支持非公平锁，当然，在大部分情况下，非公平锁是高效的选择。

## 29. 乐观锁和悲观锁的理解及如何实现，有哪些实现方式？

- 悲观锁：总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。再比如 Java 里面的同步原语 synchronized 关键字的实现也是悲观锁。
- 乐观锁：顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库提供的类似于 write\_condition 机制，其实都是提供的乐观锁。在 Java 中 java.util.concurrent.atomic 包下面的原子变量类就是使用了乐观锁的一种实现方式 CAS 实现的。

## 30. 什么是 CAS

- CAS 是 compare and swap 的缩写，即我们所说的比较交换。

- cas 是一种基于锁的操作，而且是乐观锁。在 java 中锁分为乐观锁和悲观锁。悲观锁是将资源锁住，等一个之前获得锁的线程释放锁之后，下一个线程才可以访问。而乐观锁采取了一种宽泛的态度，通过某种方式不加锁来处理资源，比如通过给记录加 version 来获取数据，性能较悲观锁有很大的提高。
- CAS 操作包含三个操作数 —— 内存位置 (V)、预期原值 (A) 和新值(B)。如果内存地址里面的值和 A 的值是一样的，那么就将内存里面的值更新成 B。CAS是通过无限循环来获取数据的，若果在第一轮循环中，a 线程获取地址里面的值被b 线程修改了，那么 a 线程需要自旋，到下次循环才有可能机会执行。

java.util.concurrent.atomic 包下的类大多是使用 CAS 操作来实现的  
(AtomicInteger,AtomicBoolean,AtomicLong)

## 31. CAS 的会产生什么问题？

- 1、ABA 问题：

比如说一个线程 one 从内存位置 V 中取出 A，这时候另一个线程 two 也从内存中取出 A，并且 two 进行了一些操作变成了 B，然后 two 又将 V 位置的数据变成 A，这时候线程 one 进行 CAS 操作发现内存中仍然是 A，然后 one 操作成功。尽管线程 one 的 CAS 操作成功，但可能存在潜藏的问题。从 Java1.5 开始 JDK 的 atomic 包里提供了一个类 AtomicStampedReference 来解决 ABA 问题。

- 2、循环时间长开销大：

对于资源竞争严重（线程冲突严重）的情况，CAS 自旋的概率会比较大，从而浪费更多的 CPU 资源，效率低于 synchronized。

- 3、只能保证一个共享变量的原子操作：

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

## 32. 什么是原子类

- java.util.concurrent.atomic包：是原子类的小工具包，支持在单个变量上解除锁的线程安全编程。原子变量类相当于一种泛化的 volatile 变量，能够支持原子的和有条件的读-改-写操作。
- 比如：AtomicInteger 表示一个int类型的值，并提供了 get 和 set 方法，这些 Volatile 类型的int 变量在读取和写入上有着相同的内存语义。它还提供了一个原子的 compareAndSet 方法（如果该方法成功执行，那么将实现与读取/写入一个 volatile 变量相同的内存效果），以及原子的添加、递增和递减等方法。AtomicInteger 表面上非常像一个扩展的 Counter 类，但在发生竞争的情况下能提供更高的可伸缩性，因为它直接利用了硬件对并发的支持。

简单来说就是原子类来实现CAS无锁模式的算法

## 33. 原子类的常用类

- AtomicBoolean
- AtomicInteger
- AtomicLong
- AtomicReference

## 34. 说一下 Atomic的原理？

- Atomic包中的类基本的特性就是在多线程环境下，当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以向自旋锁一样，继续尝试，一直等到执行成功。

## 35. 死锁与活锁的区别，死锁与饥饿的区别？

- 死锁：是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。
- 活锁：任务或者执行者没有被阻塞，由于某些条件没有满足，导致一直重复尝试，失败，尝试，失败。
- 活锁和死锁的区别在于，处于活锁的实体是在不断的改变状态，这就是所谓的“活”，而处于死锁的实体表现为等待；活锁有可能自行解开，死锁则不能。
- 饥饿：一个或者多个线程因为种种原因无法获得所需要的资源，导致一直无法执行的状态。

Java 中导致饥饿的原因：

- 1、高优先级线程吞噬所有的低优先级线程的 CPU 时间。
- 2、线程被永久堵塞在一个等待进入同步块的状态，因为其他线程总是能在它之前持续地对该同步块进行访问。
- 3、线程在等待一个本身也处于永久等待完成的对象(比如调用这个对象的 wait 方法)，因为其他线程总是被持续地获得唤醒。

# 线程池

---

## 1. 什么是线程池？

- Java中的线程池是运用场景最多的并发框架，几乎所有需要异步或并发执行任务的程序都可以使用线程池。在开发过程中，合理地使用线程池能够带来许多好处。
  - 降低资源消耗。通过重复利用已创建的线程降低线程创建和销毁造成的消耗。
  - 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。
  - 提高线程的可管理性。线程是稀缺资源，如果无限制地创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一分配、调优和监控。但是，要做到合理利用

## 2. 线程池作用？

- 线程池是为突然大量爆发的线程设计的，通过有限的几个固定线程为大量的操作服务，减少了创建和销毁线程所需的时间，从而提高效率。
- 如果一个线程所需要执行的时间非常长的话，就没必要用线程池了(不是不能作长时间操作，而是不宜。本来降低线程创建和销毁，结果你那么久我还不好控制还不如直接创建线程)，况且我们还不能控制线程池中线程的开始、挂起、和中止。

## 3. 线程池有什么优点？

- 降低资源消耗：重用存在的线程，减少对象创建销毁的开销。
- 提高响应速度。可有效的控制最大并发线程数，提高系统资源的使用率，同时避免过多资源竞争，避免堵塞。当任务到达时，任务可以不需要等的等到线程创建就能立即执行。
- 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

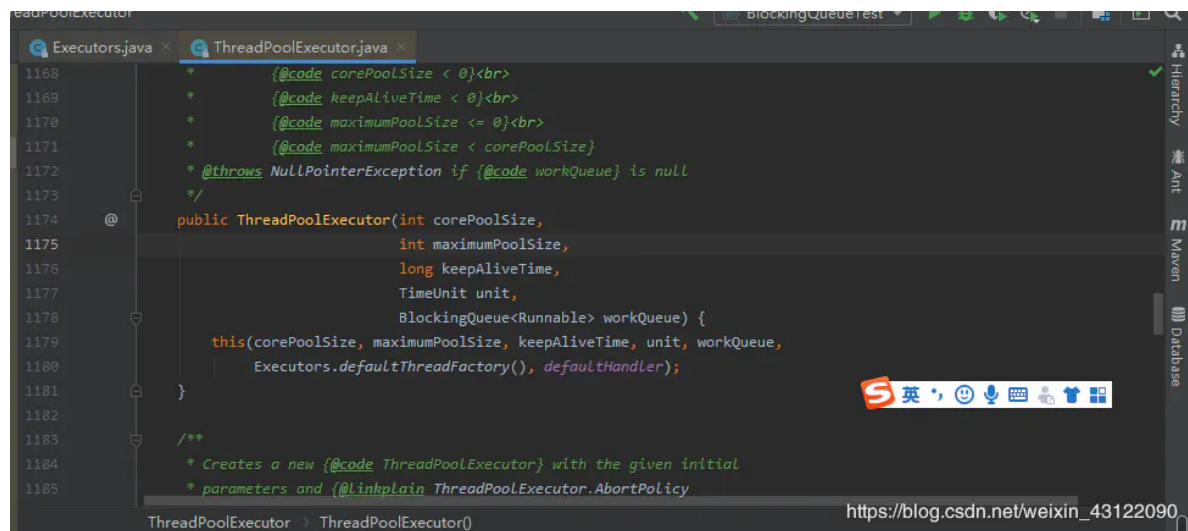
- 附加功能：提供定时执行、定期执行、单线程、并发数控制等功能。

## 4. 什么是ThreadPoolExecutor?

- **ThreadPoolExecutor就是线程池**

ThreadPoolExecutor其实也是JAVA的一个类，我们一般通过Executors工厂类的方法，通过传入不同的参数，就可以构造出适用于不同应用场景下的ThreadPoolExecutor（线程池）

构造参数图：



构造参数参数介绍：

corePoolSize 核心线程数量  
maximumPoolSize 最大线程数量  
keepAliveTime 线程保持时间，N个时间单位  
unit 时间单位（比如秒，分）  
workQueue 阻塞队列  
threadFactory 线程工厂  
handler 线程池拒绝策略

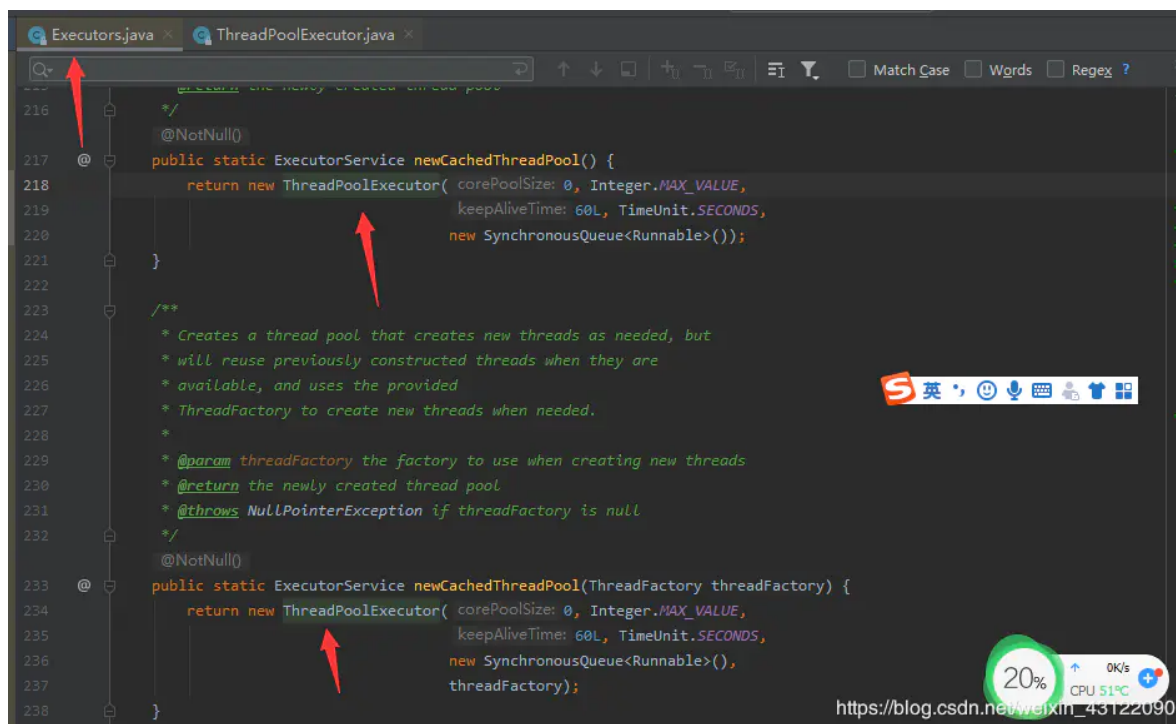
## 5. 什么是Executors?

- **Executors框架实现的就是线程池的功能。**

Executors工厂类中提供的newCachedThreadPool、newFixedThreadPool、newScheduledThreadPool、newSingleThreadExecutor等方法其实也只是ThreadPoolExecutor的构造函数参数不同而已。通过传入不同的参数，就可以构造出适用于不同应用场景下的线程池，

Executor工厂类如何创建线程池图：





```
216
217 @NotNull
218 public static ExecutorService newCachedThreadPool() {
219     return new ThreadPoolExecutor( corePoolSize: 0, Integer.MAX_VALUE,
220                                   keepAliveTime: 60L, TimeUnit.SECONDS,
221                                   new SynchronousQueue<Runnable>());
222 }
223
224 /**
225  * Creates a thread pool that creates new threads as needed, but
226  * will reuse previously constructed threads when they are
227  * available, and uses the provided
228  * ThreadFactory to create new threads when needed.
229  *
230  * @param threadFactory the factory to use when creating new threads
231  * @return the newly created thread pool
232  * @throws NullPointerException if threadFactory is null
233  */
234 @NotNull
235 public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory) {
236     return new ThreadPoolExecutor( corePoolSize: 0, Integer.MAX_VALUE,
237                                   keepAliveTime: 60L, TimeUnit.SECONDS,
238                                   new SynchronousQueue<Runnable>(),
239                                   threadFactory);
239 }
```

## 6. 线程池四种创建方式？

- Java通过Executors (jdk1.5并发包) 提供四种线程池，分别为：
  1. newCachedThreadPool创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。
  2. newFixedThreadPool 创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
  3. newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。
  4. newSingleThreadExecutor 创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

## 7. 在Java中Executor和Executors的区别？

- Executors 工具类的不同方法按照我们的需求创建了不同的线程池，来满足业务的需求。
- Executor 接口对象能执行我们的线程任务。
- ExecutorService 接口继承了 Executor 接口并进行了扩展，提供了更多的方法我们能获得任务执行的状态并且可以获取任务的返回值。
- 使用 ThreadPoolExecutor 可以创建自定义线程池。

## 8. 四种构建线程池的区别及特点？

### 1. newCachedThreadPool

- **特点：**newCachedThreadPool创建一个可缓存线程池，如果当前线程池的长度超过了处理的需要时，它可以灵活的回收空闲的线程，当需要增加时，它可以灵活的添加新的线程，而不会对池的长度作任何限制
- **缺点：**他虽然可以无线的新建线程，但是容易造成堆外内存溢出，因为它的最大值是在初始化的时候设置为 Integer.MAX\_VALUE，一般来说机器都没那么大内存给它不断使用。当然知道可能出问题的点，就可以去重写一个方法限制一下这个最大值

- **总结：**线程池为无限大，当执行第二个任务时第一个任务已经完成，会复用执行第一个任务的线程，而不用每次新建线程。
- **代码示例：**

```
package com.lijie;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TestNewCachedThreadPool {
    public static void main(String[] args) {
        // 创建无限大小线程池，由jvm自动回收
        ExecutorService newCachedThreadPool = Executors.newCachedThreadPool();
        for (int i = 0; i < 10; i++) {
            final int temp = i;
            newCachedThreadPool.execute(new Runnable() {
                public void run() {
                    try {
                        Thread.sleep(100);
                    } catch (Exception e) {
                    }
                    System.out.println(Thread.currentThread().getName() +
",i==" + temp);
                }
            });
        }
    }
}
```

## 2.newFixedThreadPool

- **特点：**创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。定长线程池的大小最好根据系统资源进行设置。
- **缺点：**线程数量是固定的，但是阻塞队列是无界队列。如果有很多请求积压，阻塞队列越来越长，容易导致OOM（超出内存空间）
- **总结：**请求的挤压一定要和分配的线程池大小匹配，定线程池的大小最好根据系统资源进行设置。如Runtime.getRuntime().availableProcessors()

Runtime.getRuntime().availableProcessors()方法是查看电脑CPU核心数量)

- **代码示例：**

```
package com.lijie;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TestNewFixedThreadPool {
    public static void main(String[] args) {
        ExecutorService newFixedThreadPool =
Executors.newFixedThreadPool(3);
        for (int i = 0; i < 10; i++) {
            final int temp = i;
            newFixedThreadPool.execute(new Runnable() {
                public void run() {
```

```

        System.out.println(Thread.currentThread().getName() +
        ",i==" + temp);
    }
    });
}
}
}

```

### 3.newScheduledThreadPool

- **特点：**创建一个固定长度的线程池，而且支持定时的以及周期性的任务执行，类似于 Timer（Timer是Java的一个定时器类）
- **缺点：**由于所有任务都是由同一个线程来调度，因此所有任务都是串行执行的，同一时间只能有一个任务在执行，前一个任务的延迟或异常都将会影响到之后的任务（比如：一个任务出错，以后的任务都无法继续）。
- **代码示例：**

```

package com.lijie;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class TestNewScheduledThreadPool {
    public static void main(String[] args) {
        //定义线程池大小为3
        ScheduledExecutorService newScheduledThreadPool =
        Executors.newScheduledThreadPool(3);
        for (int i = 0; i < 10; i++) {
            final int temp = i;
            newScheduledThreadPool.schedule(new Runnable() {
                public void run() {
                    System.out.println("i:" + temp);
                }
            }, 3, TimeUnit.SECONDS);//这里表示延迟3秒执行。
        }
    }
}

```

### 4.newSingleThreadExecutor

- **特点：**创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，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，他必须保证前一项任务执行完毕才能执行后一项。保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。
- **缺点：**缺点的话，很明显，他是单线程的，高并发业务下有点无力
- **总结：**保证所有任务按照指定顺序执行的，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
- **代码示例：**

```

package com.lijie;

import java.util.concurrent.ExecutorService;

```

```
import java.util.concurrent.Executors;

public class TestNewSingleThreadExecutor {
    public static void main(String[] args) {
        ExecutorService newSingleThreadExecutor =
        Executors.newSingleThreadExecutor();
        for (int i = 0; i < 10; i++) {
            final int index = i;
            newSingleThreadExecutor.execute(new Runnable() {
                public void run() {
                    System.out.println(Thread.currentThread().getName() + "
index:" + index);
                    try {
                        Thread.sleep(200);
                    } catch (Exception e) {
                    }
                }
            });
        }
    }
}
```

## 9. 线程池都有哪些状态？

- RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。
- SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。
- STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。
- TIDYING：所有的任务都销毁了，workCount 为 0，线程池的状态在转换为 TIDYING 状态时，会执行钩子方法 terminated()。
- TERMINATED：terminated()方法结束后，线程池的状态就会变成这个。

## 10. 线程池中 submit() 和 execute() 方法有什么区别？

- 相同点：
  - 相同点就是都可以开启线程执行池中的任务。
- 不同点：
  - 接收参数：execute()只能执行 Runnable 类型的任务。submit()可以执行 Runnable 和 Callable 类型的任务。
  - 返回值：submit()方法可以返回持有计算结果的 Future 对象，而execute()没有
  - 异常处理：submit()方便Exception处理

## 11. 什么是线程组，为什么在 Java 中不推荐使用？

- ThreadGroup 类，可以把线程归属到某一个线程组中，线程组中可以有线程对象，也可以有线程组，组中还可以有线程，这样的组织结构有点类似于树的形式。
- 线程组和线程池是两个不同的概念，他们的作用完全不同，前者是为了方便线程的管理，后者是为了管理线程的生命周期，复用线程，减少创建销毁线程的开销。
- 为什么不推荐使用线程组？因为使用有很多的安全隐患吧，没有具体追究，如果需要使用，推荐使用线程池。

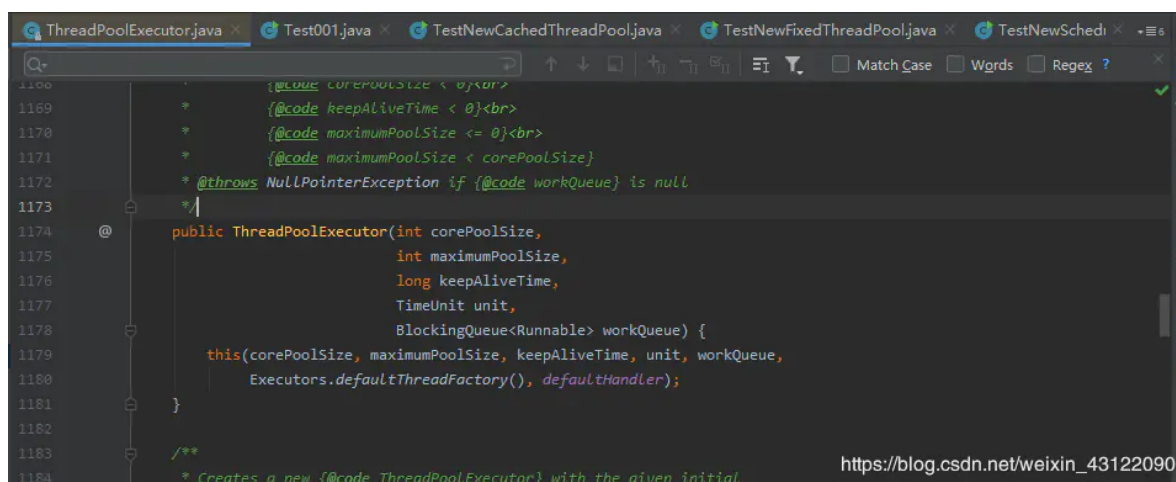
## 12. ThreadPoolExecutor饱和策略有哪些？

如果当前同时运行的线程数量达到最大线程数量并且队列也已经被放满了任时，`ThreadPoolTaskExecutor` 定义一些策略：

- `ThreadPoolExecutor.AbortPolicy`: 抛出 `RejectedExecutionException` 来拒绝新任务的处理。
- `ThreadPoolExecutor.CallerRunsPolicy`: 调用执行自己的线程运行任务。您不会任务请求。但是这种策略会降低对于新任务提交速度，影响程序的整体性能。另外，这个策略喜欢增加队列容量。如果您的应用程序可以承受此延迟并且你不能任务丢弃任何一个任务请求的话，你可以选择这个策略。
- `ThreadPoolExecutor.DiscardPolicy`: 不处理新任务，直接丢弃掉。
- `ThreadPoolExecutor.DiscardOldestPolicy`: 此策略将丢弃最早的未处理的任务请求。

### 13. 如何自定义线程线程池?

- 先看ThreadPoolExecutor（线程池）这个类的构造参数



### 构造参数参数介绍:

- `corePoolSize` 核心线程数量
- `maximumPoolSize` 最大线程数量
- `keepAliveTime` 线程保持时间，N个时间单位
- `unit` 时间单位（比如秒，分）
- `workQueue` 阻塞队列
- `threadFactory` 线程工厂
- `handler` 线程池拒绝策略

- 代码示例:

```
package com.ljjie;

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class Test001 {
    public static void main(String[] args) {
        //创建线程池
```

```

ThreadPoolExecutor executor = new ThreadPoolExecutor(1, 2, 60L,
TimeUnit.SECONDS, new ArrayBlockingQueue<>(3));
    for (int i = 1; i <= 6; i++) {
        TaskThred t1 = new TaskThred("任务" + i);
        //executor.execute(t1);是执行线程方法
    executor.execute(t1);
    }
    //executor.shutdown()不再接受新的任务，并且等待之前提交的任务都执行完再关
    闭，阻塞队列中的任务不会再执行。
    executor.shutdown();
}

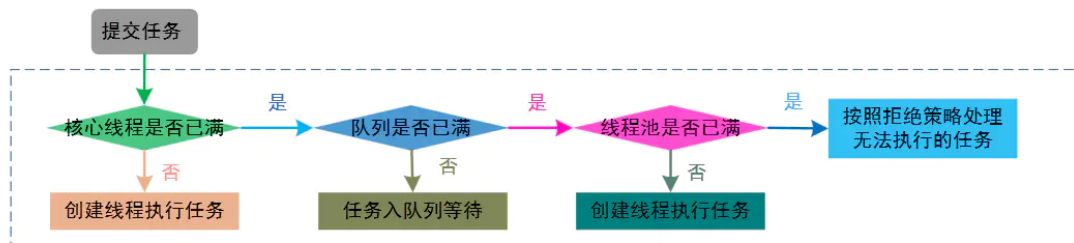
}

class TaskThred implements Runnable {
    private String taskName;

    public TaskThred(String taskName) {
        this.taskName = taskName;
    }
    public void run() {
        System.out.println(Thread.currentThread().getName() + taskName);
    }
}
}

```

## 14. 线程池的执行原理？



[https://blog.csdn.net/weixin\\_43122090](https://blog.csdn.net/weixin_43122090)

- 提交一个任务到线程池中，线程池的处理流程如下：
  - 判断线程池里的核心线程是否都在执行任务，如果不是（核心线程空闲或者还有核心线程没有被创建）则创建一个新的工作线程来执行任务。如果核心线程都在执行任务，则进入下个流程。
  - 线程池判断工作队列是否已满，如果工作队列没有满，则将新提交的任务存储在这个工作队列里。如果工作队列满了，则进入下个流程。
  - 判断线程池里的线程是否都处于工作状态，如果没有，则创建一个新的工作线程来执行任务。如果已经满了，则交给饱和策略来处理这个任务。

## 15. 如何合理分配线程池大小？

- 要合理的分配线程池的大小要根据实际情况来定，简单的来说的话就是根据CPU密集和IO密集来分配

### 什么是CPU密集

- CPU密集的意思是该任务需要大量的运算，而没有阻塞，CPU一直全速运行。



- CPU密集任务只有在真正的多核CPU上才可能得到加速(通过多线程),而在单核CPU上,无论你开几个模拟的多线程,该任务都不可能得到加速,因为CPU总的运算能力就那样。

### 什么是IO密集

- IO密集型,即该任务需要大量的IO,即大量的阻塞。在单线程上运行IO密集型的任务会导致浪费大量的CPU运算能力浪费在等待。所以在IO密集型任务中使用多线程可以大大的加速程序运行,即时在单核CPU上,这种加速主要就是利用了被浪费掉的阻塞时间。

### 分配CPU和IO密集:

1. CPU密集型时,任务可以少配置线程数,大概和机器的cpu核数相当,这样可以使得每个线程都在执行任务
2. IO密集型时,大部分线程都阻塞,故需要多配置线程数,  $2 * \text{cpu核数}$

### 精确来说的话的话:

- 从以下几个角度分析任务的特性:
  - 任务的性质: CPU密集型任务、IO密集型任务、混合型任务。
  - 任务的优先级: 高、中、低。
  - 任务的执行时间: 长、中、短。
  - 任务的依赖性: 是否依赖其他系统资源,如数据库连接等。

### 可以得出一个结论:

- 线程等待时间比CPU执行时间比例越高,需要越多线程。
- 线程CPU执行时间比等待时间比例越高,需要越少线程。

## 并发容器

### 1. 你经常使用什么并发容器,为什么?

- 答: Vector、ConcurrentHashMap、HasTable

- 一般软件开发中容器用的最多的就是HashMap、ArrayList、LinkedList，等等
- 但是在多线程开发中就不能乱用容器，如果使用了未加锁（非同步）的集合，你的数据就会非常的混乱。由此在多线程开发中需要使用的容器必须是加锁（同步）的容器。

## 2. 什么是Vector

- Vector与ArrayList一样，也是通过数组实现的，不同的是它支持线程的同步，即某一时刻只有一个线程能够写Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，访问它比访问ArrayList慢很多

(ArrayList是最常用的List实现类，内部是通过数组实现的，它允许对元素进行快速随机访问。当从ArrayList的中间位置插入或者删除元素时，需要对数组进行复制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。ArrayList的缺点是每个元素之间不能有间隔。)

## 3. ArrayList和Vector有什么不同之处？

- Vector方法带上了synchronized关键字，是线程同步的

### 1. ArrayList添加方法源码

```
public boolean add(E e) {  
    modCount++;  
    add(e, elementData, size);  
    return true;  
}
```

### 2. Vector添加源码（加锁了synchronized关键字）

```
public synchronized boolean add(E e) {  
    modCount++;  
    add(e, elementData, elementCount);  
    return true;  
}
```

## 4. 为什么HashTable是线程安全的？

- 因为HasTable的内部方法都被synchronized修饰了，所以是线程安全的。其他的都和HashMap一样

### 1. HashMap添加方法的源码

```
public V put(K key, V value) { return putVal(hash(key), key, value, onlyIfAbsent: false, evict: true); }  
  
/**  
 * Implements Map.put and related methods.  
 */
```

### 2. HashTable添加方法的源码

```

public synchronized V put(K key, V value) {
    // Make sure the value is not null
    if (value == null) {
        throw new NullPointerException();
    }
}

```

## 5. 用过ConcurrentHashMap，讲一下他和HashTable的不同之处？

- ConcurrentHashMap是Java5中支持高并发、高吞吐量的线程安全HashMap实现。它由Segment数组结构和HashEntry数组结构组成。Segment数组在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键-值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构；一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素；每个Segment守护着一个HashEntry数组里的元素，当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。
- 看不懂??? 很正常，我也看不懂
- 总结：
  - HashTable就是实现了HashMap加上了synchronized，而ConcurrentHashMap底层采用分段的数组+链表实现，线程安全
  - ConcurrentHashMap通过把整个Map分为N个Segment，可以提供相同的线程安全，但是效率提升N倍，默认提升16倍。
  - 并且读操作不加锁，由于HashEntry的value变量是volatile的，也能保证读取到最新的值。
  - Hashtables的synchronized是针对整张Hash表的，即每次锁住整张表让线程独占，ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术
  - 扩容：段内扩容（段内元素超过该段对应Entry数组长度的75%触发扩容，不会对整个Map进行扩容），插入前检测需不需要扩容，有效避免无效扩容

## 6. Collections.synchronized \* 是什么？

注意：\* 号代表后面是还有内容的

- 此方法是干什么的呢，他完全全的可以把List、Map、Set接口底下的集合变成线程安全的集合
- Collections.synchronized \*：原理是什么，我猜的话是代理模式

```

public class Test {
    public static void main(String[] args) throws InterruptedException {
        List<String> list = Collections.synchronizedList(new ArrayList<>());
        Collections.synchronized
    }
}

```

synchronizedList(List<T> list) List<T>  
 synchronizedCollection(Collection<T> c) Collection<T>  
 synchronizedMap(Map<K, V> m) Map<K, V>  
 synchronizedNavigableMap(NavigableMap<K, V> m) NavigableMap<K, V>  
 synchronizedNavigableSet(NavigableSet<T> s) NavigableSet<T>  
 synchronizedSet(Set<T> s) Set<T>  
 synchronizedSortedMap(SortedMap<K, V> m) SortedMap<K, V>  
 synchronizedSortedSet(SortedSet<T> s) SortedSet<T>

Press Enter to insert Tab to replace Next Tip

[https://blog.csdn.net/weixin\\_43122090](https://blog.csdn.net/weixin_43122090)

## 7. Java 中 ConcurrentHashMap 的并发度是什么？

- ConcurrentHashMap 把实际 map 划分成若干部分来实现它的可扩展性和线程安全。这种划分是使用并发度获得的，它是 ConcurrentHashMap 类构造函数的一个可选参数，默认值为 16，这样在多线程情况下就能避免争用。
- 在 JDK8 后，它摒弃了 Segment（锁段）的概念，而是启用了一种全新的方式实现，利用 CAS 算法。同时加入了更多的辅助变量来提高并发度，具体内容还是查看源码吧。

## 8. 什么是并发容器的实现？

- 何为同步容器：可以简单地理解为通过 synchronized 来实现同步的容器，如果有多个线程调用同步容器的方法，它们将会串行执行。比如 Vector，Hashtable，以及 Collections.synchronizedSet，synchronizedList 等方法返回的容器。可以通过查看 Vector，Hashtable 等这些同步容器的实现代码，可以看到这些容器实现线程安全的方式就是将它们的状态封装起来，并在需要同步的方法上加上关键字 synchronized。
- 并发容器使用了与同步容器完全不同的加锁策略来提供更高的并发性和伸缩性，例如在 ConcurrentHashMap 中采用了一种粒度更细的加锁机制，可以称为分段锁，在这种锁机制下，允许任意数量的读线程并发地访问 map，并且执行读操作的线程和写操作的线程也可以并发的访问 map，同时允许一定数量的写操作线程并发地修改 map，所以它可以在并发环境下实现更高的吞吐量。

## 9. Java 中的同步集合与并发集合有什么区别？

- 同步集合与并发集合都为多线程和并发提供了合适的线程安全的集合，不过并发集合的可扩展性更高。在 Java1.5 之前程序员们只有同步集合来用且在多线程并发的时候会导致争用，阻碍了系统的扩展性。Java5 介绍了并发集合像 ConcurrentHashMap，不仅提供线程安全还用锁分离和内部分区等现代技术提高了可扩展性。

## 10. SynchronizedMap 和 ConcurrentHashMap 有什么区别？

- SynchronizedMap 一次锁住整张表来保证线程安全，所以每次只能有一个线程来访问 map。
- ConcurrentHashMap 使用分段锁来保证在多线程下的性能。
- ConcurrentHashMap 中则是一次锁住一个桶。ConcurrentHashMap 默认将 hash 表分为 16 个桶，诸如 get，put，remove 等常用操作只锁当前需要用到的桶。
- 这样，原来只能一个线程进入，现在却能同时有 16 个写线程执行，并发性能的提升是显而易见的。
- 另外 ConcurrentHashMap 使用了一种不同的迭代方式。在这种迭代方式中，当 iterator 被创建后集合再发生改变就不再是抛出 ConcurrentModificationException，取而代之的是在改变时 new 新的数据从而不影响原有的数据，iterator 完成后再将头指针替换为新的数据，这样 iterator 线程可以使用原来老的数据，而写线程也可以并发的完成改变。

## 11. CopyOnWriteArrayList 是什么？

- CopyOnWriteArrayList 是一个并发容器。有很多人称它是线程安全的，我认为这句话不严谨，缺少一个前提条件，那就是非复合场景下操作它是线程安全的。
- CopyOnWriteArrayList(免锁容器)的好处之一是当多个迭代器同时遍历和修改这个列表时，不会抛出 ConcurrentModificationException。在 CopyOnWriteArrayList 中，写入将导致创建整个底层数组的副本，而源数组将保留在原地，使得复制的数组在被修改时，读取操作可以安全地执行。

## 12. CopyOnWriteArrayList 的使用场景？

- 合适读多写少的场景。

## 13. CopyOnWriteArrayList 的缺点？

- 由于写操作的时候，需要拷贝数组，会消耗内存，如果原数组的内容比较多的情况下，可能导致 young gc 或者 full gc。
- 不能用于实时读的场景，像拷贝数组、新增元素都需要时间，所以调用一个 set 操作后，读取到数据可能还是旧的，虽然 CopyOnWriteArrayList 能做到最终一致性，但是还是没法满足实时性要求。
- 由于实际使用中可能没法保证 CopyOnWriteArrayList 到底要放置多少数据，万一数据稍微有点多，每次 add/set 都要重新复制数组，这个代价实在太高昂了。在高性能的互联网应用中，这种操作分分钟引起故障。

## 14. CopyOnWriteArrayList 的设计思想？

- 读写分离，读和写分开
- 最终一致性
- 使用另外开辟空间的思路，来解决并发冲突

# 并发队列

## 1. 什么是并发队列：

- 消息队列很多人知道：消息队列是分布式系统中重要的组件，是系统与系统直接的通信
- 并发队列是什么：并发队列多个线程以有次序共享数据的重要组件

## 2. 并发队列和并发集合的区别：

那就有可能要说了，我们并发集合不是也可以实现多线程之间的数据共享吗，其实也是有区别的：

- 队列遵循“先进先出”的规则，可以想象成排队检票，队列一般用来解决大数据量采集处理和显示的。
- 并发集合就是在多个线程中共享数据的

## 3. 怎么判断并发队列是阻塞队列还是非阻塞队列

- 在并发队列上JDK提供了Queue接口，一个是以Queue接口下的BlockingQueue接口为代表的阻塞队列，另一个是高性能（无堵塞）队列。

## 4. 阻塞队列和非阻塞队列区别

- 当队列阻塞队列为空的时，从队列中获取元素的操作将会被阻塞。
- 或者当阻塞队列是满时，往队列里添加元素的操作会被阻塞。
- 或者试图从空的阻塞队列中获取元素的线程将会被阻塞，直到其他的线程往空的队列插入新的元素。

- 试图往已满的阻塞队列中添加新元素的线程同样也会被阻塞，直到其他的线程使队列重新变得空闲起来

## 5. 常用并发队列的介绍：

### 1. 非堵塞队列：

#### 1. ArrayDeque, (数组双端队列)

ArrayDeque (非堵塞队列) 是JDK容器中的一个双端队列实现，内部使用数组进行元素存储，不允许存储null值，可以高效的进行元素查找和尾部插入取出，是用作队列、双端队列、栈的绝佳选择，性能比LinkedList还要好。

#### 2. PriorityQueue, (优先级队列)

PriorityQueue (非堵塞队列) 一个基于优先级的无界优先级队列。优先级队列的元素按照其自然顺序进行排序，或者根据构造队列时提供的 Comparator 进行排序，具体取决于所使用的构造方法。该队列不允许使用 null 元素也不允许插入不可比较的对象

#### 3. ConcurrentLinkedQueue, (基于链表的并发队列)

ConcurrentLinkedQueue (非堵塞队列) : 是一个适用于高并发场景下的队列，通过无锁的方式，实现了高并发状态下的高性能。ConcurrentLinkedQueue的性能要好于BlockingQueue接口，它是一个基于链接节点的无界线程安全队列。该队列的元素遵循先进先出的原则。该队列不允许null元素。

### 4. 堵塞队列：

#### 1. DelayQueue, (基于时间优先级的队列，延期阻塞队列)

DelayQueue是一个没有边界BlockingQueue实现，加入其中的元素必需实现Delayed接口。当生产者线程调用put之类的方法加入元素时，会触发Delayed接口中的compareTo方法进行排序，也就是说队列中元素的顺序是按到期时间排序的，而非它们进入队列的顺序。排在队列头部的元素是最早到期的，越往后到期时间越晚。

#### 5. ArrayBlockingQueue, (基于数组的并发阻塞队列)

ArrayBlockingQueue是一个有边界的阻塞队列，它的内部实现是一个数组。有边界的意思是它的容量是有限的，我们必须在初始化时指定它的容量大小，容量大小一旦指定就不可改变。ArrayBlockingQueue是以先进先出的方式存储数据

#### 6. LinkedBlockingQueue, (基于链表的FIFO阻塞队列)

LinkedBlockingQueue阻塞队列大小的配置是可选的，如果我们初始化时指定一个大小，它就是有边界的，如果不指定，它就是无边界的。说是无边界，其实是采用了默认大小为Integer.MAX\_VALUE的容量。它的内部实现是一个链表。

#### 7. LinkedBlockingDeque, (基于链表的FIFO双端阻塞队列)

LinkedBlockingDeque是一个由链表结构组成的双向阻塞队列，即可以从队列的两端插入和移除元素。双向队列因为多了一个操作队列的入口，在多线程同时入队时，也就减少了一半的竞争。

相比于其他阻塞队列，LinkedBlockingDeque多了addFirst、addLast、peekFirst、peekLast等方法，以first结尾的方法，表示插入、获取移除双端队列的第一个元素。以last结尾的方法，表示插入、获取移除双端队列的最后一个元素。

LinkedBlockingDeque是可选容量的，在初始化时可以设置容量防止其过度膨胀，如果不设置，默认容量大小为Integer.MAX\_VALUE。

#### 5. PriorityBlockingQueue, (带优先级的无界阻塞队列)

priorityBlockingQueue是一个无界队列，它没有限制，在内存允许的情况下可以无限添加元素；它又是具有优先级的队列，是通过构造函数传入的对象来判断，传入的对象必须实现comparable接口。

6. SynchronousQueue（并发同步阻塞队列）

SynchronousQueue是一个内部只能包含一个元素的队列。插入元素到队列的线程被阻塞，直到另一个线程从队列中获取了队列中存储的元素。同样，如果线程尝试获取元素并且当前不存在任何元素，则该线程将被阻塞，直到线程将元素插入队列。

将这个类称为队列有点夸大其词。这更像是一个点。

并发队列的常用方法

不管是那种队列，是那个类，当是他们使用的方法都是差不多的

| 方法名                               | 描述                                                               |
|-----------------------------------|------------------------------------------------------------------|
| add()                             | 在不超出队列长度的情况下插入元素，可以立即执行，成功返回true，如果队列满了就抛出异常。                    |
| offer()                           | 在不超出队列长度的情况下插入元素的时候则可以立即在队列的尾部插入指定元素,成功时返回true，如果此队列已满，则返回false。 |
| put()                             | 插入元素的时候，如果队列满了就进行等待，直到队列可用。                                      |
| take()                            | 从队列中获取值，如果队列中没有值，线程会一直阻塞，直到队列中有值，并且该方法取得了该值。                     |
| poll(long timeout, TimeUnit unit) | 在给定的时间里，从队列中获取值，如果没有取到会抛出异常。                                     |
| remainingCapacity()               | 获取队列中剩余的空间。                                                      |
| remove(Object o)                  | 从队列中移除指定的值。                                                      |
| contains(Object o)                | 判断队列中是否拥有该值。                                                     |
| drainTo(Collection c)             | 将队列中值，全部移除，并发设置到给定的集合中。                                          |

并发工具类

1. 常用的并发工具类有哪些？

- CountDownLatch
  - CountDownLatch 类位于java.util.concurrent包下，利用它可以实现类似计数器的功能。比如有一个任务A，它要等待其他3个任务执行完毕之后才能执行，此时就可以利用CountDownLatch来实现这种功能了。
- CyclicBarrier (回环栅栏) CyclicBarrier它的作用就是会让所有线程都等待完成后才会继续下一步行动。
  - CyclicBarrier初始化时规定一个数目，然后计算调用了CyclicBarrier.await()进入等待的线程数。当线程数达到了这个数目时，所有进入等待状态的线程被唤醒并继续。



CyclicBarrier初始时还可带一个Runnable的参数，此Runnable任务在CyclicBarrier的数目达到后，所有其它线程被唤醒前被执行。

- Semaphore (信号量) Semaphore 是 synchronized 的加强版，作用是控制线程的并发数量（允许自定义多少线程同时访问）。就这一点而言，单纯的synchronized 关键字是实现不了的。