**一、多线程**

**说明下线程的状态**

java中的线程一共有 5 种状态。

1. **NEW**：这种情况指的是，通过 New 关键字创建了 Thread 类（或其子类）的对象
2. **RUNNABLE**：这种情况指的是 Thread 类的对象调用了 start() 方法，这时的线程就等待时间片轮转到自己这，以便获得 CPU；第二种情况是线程在处于 RUNNING 状态时并没有运行完自己的 run 方法，时间片用完之后回到 RUNNABLE 状态；还有种情况就是处于 BLOCKED 状态的线程结束了当前的 BLOCKED 状态之后重新回到 RUNNABLE 状态。
3. **RUNNING**：这时的线程正在被 CPU 执行任务。
4. **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。阻塞的情况分三种：
   * 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。
   * 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。
   * 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。
5. **DEAD**：线程执行完了或者因异常退出了 run() 方法，该线程结束生命周期。

**sleep 和 wait 的区别**

**区别一**：

==sleep 是 Thread 类的方法，是线程用来控制自身流程的==，比如有一个要报时的线程，每一秒中打印出一个时间，那么我就需要在print方法前面加上一个sleep让自己每隔一秒执行一次。就像个闹钟一样。

==wait 是 Object 类的方法，用来线程间的通信==，这个方法会使当前拥有该对象锁的进程等待直到其他线程调用 notify 方法时再醒来，不过你也可以给他指定一个时间，自动醒来。这个方法主要是用走不同线程之间的调度的。

**区别二**：

调用 sleep 方法不会释放锁，调用 wait 方法会释放当前线程的锁。

**区别三**：

在使用域上， wait 必须在同步方法内使用，即必须在获取锁之后调用，否则会报出 IllegalMonitorStateException 异常。而 sleep 可以在任意地方使用。

**synchronized 的实现原理**

synchronized 代码块是通过 monitorenter 和 monitorexit 指令实现的。synchronized 方法虽然在 vm 字节码层面并没有任何特别的指令来实现被 synchronized 修饰的方法，而是在 Class 文件的方法表中将该方法的 access\_flags 字段中的 synchronized 标志位置1，表示该方法是同步方法。锁的实现有偏向锁、轻量级锁和重量级锁，其中偏向锁和轻量级锁是 JDK 针对锁的优化措施。在多线程的竞争下锁会升级，依次从偏向锁 -> 轻量级锁 -> 重量级锁，这里的锁只能升级但不能降级。在 Java 对象头中的 Mark Word 中存储了关于锁的标志位，其中：无锁和偏向锁为 00， 轻量级锁为 01，重量级锁为 10。

* **引入偏向锁主要目的是**：为了在无多线程竞争的情况下尽量减少不必要的轻量级锁执行路径(在无竞争的情况下把整个同步都消除掉，连 CAS 操作都不做了)。
* **引入轻量级锁的主要目的是**：在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗(在无竞争的情况下使用 CAS 操作去消除同步使用的互斥量)。
* **重量级锁**通过对象内部的监视器（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。

**锁的优缺点对比如下**

| **锁** | **优点** | **缺点** | **适用场景** |
| --- | --- | --- | --- |
| 偏向锁 | 加锁和解锁不需要额外的消耗，和执行非同步方法比仅存在纳秒级的差距。 | 如果线程间存在锁竞争，会带来额外的锁撤销的消耗。 | 适用于只有一个线程访问同步块场景。 |
| 轻量级锁 | 竞争的线程不会阻塞，提高了程序的响应速度。 | 如果始终得不到锁竞争的线程使用自旋会消耗CPU。 | 追求响应时间。同步块执行速度非常快。 |
| 重量级锁 | 线程竞争不使用自旋，不会消耗CPU。 | 线程阻塞，响应时间缓慢。 | 追求吞吐量。同步块执行速度较长。 |

此外，jdk 1.6 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、锁粗化等技术来减少锁操作的开销。

**自旋锁**

所谓自旋锁，就是让该线程等待一段时间，不会被立即挂起，看持有锁的线程是否会很快释放锁。

**适应自旋锁**

自适应就意味着自旋的次数不再是固定的，它是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。==对于同一个锁，上一个线程如果自旋成功了，那么下次自旋的次数会更加多==，因为虚拟机认为既然上次成功了，那么此次自旋也很有可能会再次成功，那么它就会允许自旋等待持续的次数更多。==反之，如果对于某个锁，很少有自旋能够成功的，那么在以后要获得这个锁的时候自旋的次数会减少甚至省略掉自旋过程，以免浪费处理器资源==。

**锁消除**

在有些情况下，JVM 检测到不可能存在共享数据竞争，这时 JVM 会对这些同步锁进行锁消除。锁消除的依据是逃逸分析的数据支持。

**锁粗化**

将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁，来减少频繁的加锁和释放锁带来的性能损耗。

**volatile 关键字**

volatile 关键字，具有两个特性：1. 内存的可见性，2. 禁止指令重排序优化。

**内存可见性是指**：被 volatile 关键字修饰的变量，当线程要对这个变量执行的写操作，都不会写入本地缓存，而是直接刷入主内存中。当线程读取被 volatile 关键字修饰的变量时，也是直接从主内存中读取。（简单的说，一个线程修改的状态对另一个线程是可见的）。**注意**：volatile 不能保证原子性。

**禁止指令重排序优化**：有volatile修饰的变量，赋值后多执行了一个 “load addl $0x0, (%esp)” 操作，这个操作相当于一个内存屏障，保证指令重排序时不会把后面的指令重排序到内存屏障之前的位置。

**详细说明指令重排序和内存屏障**

说到指令重排序，那么就要说到 as-if-serial 语义和 happens-before 规则了。

**as-if-serial 的语义是指**：不管怎么重排序，单线程程序的执行结果不能被改变。编译器、runtime和处理器都必须遵守“as-if-serial”语义。为了遵守as-if-serial语义，编译器和处理器不会对存在数据依赖关系的操作做重排序，因为这种重排序会改变执行结果。但是，如果操作之间不存在数据依赖关系，这些操作就可能被编译器和处理器重排序。

**happens-before 规则主要有一下 8 个**：

★1. 程序次序规则(Program Order Rule)：在一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作。准确地说应该是控制流顺序而不是代码顺序，因为要考虑分支、循环等结构。

★2. 监视器锁定规则(Monitor Lock Rule)：一个 unlock 操作先行发生于后面对同一个对象锁的lock操作。这里强调的是同一个锁，而“后面”指的是时间上的先后顺序，如发生在其他线程中的 lock 操作。

★3. volatile变量规则(Volatile Variable Rule):对一个 volatile 变量的写操作发生于后面对这个变量的读操作，这里的“后面”也指的是时间上的先后顺序。

1. 线程启动规则(Thread Start Rule)：Thread 独享的 start() 方法先行于此线程的每一个动作。
2. 线程终止规则(Thread Termination Rule)：线程中的每个操作都先行发生于对此线程的终止检测，我们可以通过 Thread.join() 方法结束、Thread.isAlive() 的返回值检测到线程已经终止执行。
3. 线程中断规则(Thread Interruption Rule)：对线程 interrupte() 方法的调用优先于被中断线程的代码检测到中断事件的发生，可以通过 Thread.interrupted() 方法检测线程是否已中断。
4. 对象终结原则(Finalizer Rule)：一个对象的初始化完成(构造函数执行结束)先行发生于它的 finalize() 方法的开始。

★8. 传递性(Transitivity)：如果操作A先行发生于操作B，操作B先行发生于操作C，那就可以得出操作A先行发生于操作C的结论。

我认为比较重要的规则是：程序次序规则、监视器锁定规则、volatile 变量规则和传递性。

除此之外，Java 内存模型对 volatile 和 final 的语义做了扩展。对 volatile 语义的扩展保证了 volatile 变量在一些情况下不会重排序，volatile 的 64 位变量 double 和 long 的读取和赋值操作都是原子的。

**对于基本类型 final 域，编译器和处理器要遵守两个重排序规则：**

1. 在构造函数内对一个final域的写入，与随后把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。（StoreStore屏障）
2. 初次读一个包含final域的对象的引用，与随后初次读这个final域，这两个操作之间不能重排序。（LoadLoad屏障）

**对于引用类型**，写final域的重排序规则对编译器和处理器增加了如下约束：

1. 在构造函数内对一个final引用的对象的成员域的写入，与随后在构造函数外把这个被构造对象的引用赋值给一个引用变量，这两个操作之间不能重排序。

**关于内存屏障**

内存屏障，是一种CPU指令，用于控制特定条件下的重排序和内存可见性问题。Java编译器也会根据内存屏障的规则禁止重排序。

内存屏障可以被分为以下几种类型

1. **LoadLoad屏障**：对于这样的语句Load1; LoadLoad; Load2，在Load2及后续读取操作要读取的数据被访问前，保证Load1要读取的数据被读取完毕。
2. **StoreStore屏障**：对于这样的语句Store1; StoreStore; Store2，在Store2及后续写入操作执行前，保证Store1的写入操作对其它处理器可见。
3. **LoadStore屏障**：对于这样的语句Load1; LoadStore; Store2，在Store2及后续写入操作被刷出前，保证Load1要读取的数据被读取完毕。
4. **StoreLoad屏障**：对于这样的语句Store1; StoreLoad; Load2，在Load2及后续所有读取操作执行前，保证Store1的写入对所有处理器可见。**它的开销是四种屏障中最大的。 在大多数处理器的实现中，这个屏障是个万能屏障，兼具其它三种内存屏障的功能。**

**什么是 CAS?**

CAS,compare and swap的缩写，中文翻译成比较并交换。CAS指令在Intel CPU上称为CMPXCHG指令，它的作用是将指定内存地址的内容与所给的某个值相比，如果相等，则将其内容替换为指令中提供的新值，如果不相等，则更新失败。

从内存领域来说这是乐观锁，因为它在对共享变量更新之前会先比较当前值是否与更新前的值一致，如果是，则更新，如果不是，则无限循环执行(称为自旋)，直到当前值与更新前的值一致为止，才执行更新。

CAS 有 3 个操作数，内存值 V，旧的预期值 A，要修改的新值 B。当且仅当预期值 A 和内存值 V 相同时，将内存值 V 修改为 B，否则什么都不做。

CAS虽然很高效的解决原子操作，但是CAS仍然存在三大问题。ABA问题，循环时间长开销大和只能保证一个共享变量的原子操作

1. **ABA问题**。因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，
2. 那么使用CAS进行检查时会发现它的值没有发生变化，但是实际上却变化了。ABA问题的解决思路就是使用版本号。
3. 在变量前面追加上版本号，每次变量更新的时候把版本号加一，那么A－B－A 就会变成1A-2B－3A。
4. 从Java1.5开始JDK的atomic包里提供了一个类 **AtomicStampedReference** 来解决ABA问题。
5. 这个类的compareAndSet方法作用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。

1. **循环时间长开销大**。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。
2. **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，
3. 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，可以把多个共享变量合并成一个共享变量来操作。从Java1.5开始JDK提供了AtomicReference类来保证引用对象之间的原子性，可以把多个变量放在一个对象里来进行CAS操作。

**AQS 的实现原理（JUC 的并发包中的很多类都是基于这个模板方法的，很重要）**

AbstractQueuedSynchronizer(简称AQS)，队列同步器，是用来构建锁或者其他同步组建的基础框架。AQS的核心思想是基于volatile int state这样的volatile变量，配合Unsafe工具对其原子性的操作来实现对当前锁状态进行修改。同步器内部依赖一个FIFO的双向队列来完成资源获取线程的排队工作。

AQS 的锁在内部实现上分为独占模式和共享模式两种。

**独占锁**： 锁在一个时间点只能被一个线程占有。ReentrantLock 和 ReentrantReadWriteLock.Writelock 是独占锁。

**共享锁**：同一个时候能够被多个线程获取的锁，能被共享的锁。JUC包中ReentrantReadWriteLock.ReadLock，CyclicBarrier，CountDownLatch和Semaphore都是共享锁。

**AQS 独占模式获取锁的过程**：

1. 调用自定义同步器的tryAcquire()尝试直接去获取资源，如果成功则直接返回
2. 没成功，则addWaiter()将该线程封装成 Node 对象并加入等待队列的尾部，并标记为独占模式
3. acquireQueued()使线程在等待队列中休息，有机会时（轮到自己，会被unpark()）会去尝试获取资源。获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false
4. 如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上

**AQS 独占模式释放锁的过程**：

1. release() 根据 tryRelease() 的返回值来判断该线程是否已经完成释放掉资源了，如果已经彻底释放资源(state=0)，要返回true，否则返回false。
2. 如果已经彻底释放资源，则调用 unparkSuccessor 方法，唤醒等待队列里的下一个线程。

**AQS 共享模式获取锁的过程**：

1. 调用子类实现的 tryAcquireShared() 方法尝试获取资源，当前线程获取资源成功后，如果还有剩余资源，那么还会唤醒后面的线程来尝试获取资源。
2. 失败则通过doAcquireShared()进入等待队列park()，直到被unpark()/interrupt()并成功获取到资源才返回。整个等待过程也是忽略中断的。

**AQS 共享模式释放锁的过程**：

1. 调用子类实现的 tryReleaseShared() 方法释放资源，如果成功释放资源，那么就调用 doReleaseShared() 方法唤醒后面的线程。

**ReentrantLock 实现原理**

ReentrantLock 是基于 AQS 实现的独占锁。内部分为公平锁和非公平锁，默认使用的是非公平锁。

**公平锁是指**：线程获取锁的顺序和调用lock的顺序一样，FIFO。

**非公平锁是指**：线程获取锁的顺序和调用lock的顺序无关，全凭运气。 ReentrantLock 默认使用非公平锁是基于性能考虑，公平锁为了保证线程规规矩矩地排队，需要增加阻塞和唤醒的时间开销。如果直接插队获取非公平锁，跳过了对队列的处理，速度会更快。

公平锁和非公平锁在释放锁的步骤上没有区别，只是在加锁的时候有区别，区别如下：

**非公平锁加锁的步骤**：

忽视队列前面的等待线程，上来直接基于CAS尝试将state（锁数量）从0设置为1

A、如果设置成功，设置当前线程为独占锁的线程；

B、如果设置失败，还会再获取一次锁数量，

B1、如果锁数量为0，再基于CAS尝试将state（锁数量）从0设置为1一次，如果设置成功，设置当前线程为独占锁的线程；

B2、如果锁数量不为0或者上边的尝试又失败了，查看当前线程是不是已经是独占锁的线程了，如果是，则将当前的锁数量+1；如果不是，则将该线程封装在一个Node内，并加入到等待队列中去。等待被其前一个线程节点唤醒。

**公平锁加锁的步骤**：

A1、获取一次锁数量

B1、如果锁数量为0，如果当前线程是等待队列中的头节点，基于CAS尝试将state（锁数量）从0设置为1一次，如果设置成功，设置当前线程为独占锁的线程；

B2、如果锁数量不为0或者当前线程不是等待队列中的头节点或者上边的尝试又失败了，查看当前线程是不是已经是独占锁的线程了，如果是，则将当前的锁数量+1；如果不是，则将该线程封装在一个Node内，并加入到等待队列中去。等待被其前一个线程节点唤醒。

**线程池的实现原理**

在 Java 中一共有 5 种线程池，他们分别是：CachedThreadPool，FixedThreadPool，SingleThreadExecutor，ScheduleThreadPool，ScheduledThreadPoolExecutor。它们分别是通过 Executors 的静态方法创建出来的。而他们底层是通过 ThreadPoolExecutor 类创建出来的。创建线程池时会传入以下参数，他们分别是：

1. corePoolSize：核心线程池的大小，在线程池被创建之后，其实里面是没有线程的。（当然，调用 prestartAllCoreThreads() 或者 prestartCoreThread() 方法会预创建线程，而不用等着任务的到来）。当有任务进来的时候，才会创建线程。当线程池中的线程数量达到corePoolSize之后，就把任务放到缓存队列当中。（就是 workQueue ）。
2. maximumPoolSize：最大线程数量是多少。它标志着这个线程池的最大线程数量。如果没有最大数量，当创建的线程数量达到了 某个极限值，到最后内存肯定就爆掉了。
3. keepAliveTime：当线程没有任务时，最多保持的时间，超过这个时间就被终止了,默认值 60 秒。默认情况下，只有线程池中线程数量大于 corePoolSize 时，keepAliveTime 值才会起作用。也就说，只有在线程池线程数量超出 corePoolSize 了。我们才会把超时的空闲线程给停止掉。否则就保持线程池中有 corePoolSize 个线程就可以了。
4. Unit：参数keepAliveTime的时间单位，就是 TimeUnit类当中的几个属性。
5. workQueue：用来存储待执行任务的队列，不同的线程池它的队列实现方式不同（因为这关系到排队策略的问题）比如有以下几种:
   * ArrayBlockingQueue：基于数组的队列，创建时需要指定大小。
   * LinkedBlockingQueue：基于链表的队列，如果没有指定大小，则默认值是 Integer.MAX\_VALUE。（newFixedThreadPool和newSingleThreadExecutor使用的就是这种队列），吞吐量通常要高于ArrayBlockingQuene。
   * SynchronousQueue：一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQuene（newCachedThreadPool使用的就是这种队列）。
6. threadFactory:线程工厂，用来创建线程。通过自定义的线程工厂可以给每个新建的线程设置一个具有识别度的线程名。
7. Handler：拒绝执行任务时的策略，一般来讲有以下四种策略：
   * ThreadPoolExecutor.AbortPolicy 丢弃任务，并抛出 RejectedExecutionException 异常。
   * ThreadPoolExecutor.CallerRunsPolicy：该任务被线程池拒绝，由调用 execute 方法的线程执行该任务。
   * ThreadPoolExecutor.DiscardOldestPolicy ： 抛弃队列最前面的任务，然后重新尝试执行任务。
   * ThreadPoolExecutor.DiscardPolicy，丢弃任务，不过也不抛出异常。

**几种线程池的比较**

**2.1 CachedThreadPool**

**优点**：

工作线程的创建数量几乎没有限制(其实也有限制的,数目为Interger. MAX\_VALUE), 这样可灵活的往线程池中添加线程。

如果长时间没有往线程池中提交任务，即如果工作线程空闲了指定的时间(默认为1分钟)，则该工作线程将自动终止。终止后，如果你又提交了新的任务，则线程池重新创建一个工作线程。

**缺点**：

在使用CachedThreadPool时，一定要注意控制任务的数量，否则，由于大量线程同时运行，很有会造成系统瘫痪。

**2.2 FixedThreadPool**

创建一个指定工作线程数量的线程池。每当提交一个任务就创建一个工作线程，如果工作线程数量达到线程池初始的最大数，则将提交的任务存入到池队列中。定长线程池的大小最好根据系统资源进行设置如Runtime.getRuntime().availableProcessors()

**优点**：

FixedThreadPool是一个典型且优秀的线程池，它具有线程池提高程序效率和节省创建线程时所耗的开销的优点。

**缺点**：

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。

**2.3 SingleThreadExecutor**

创建一个单线程化的Executor，即只创建唯一的工作者线程来执行任务，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。如果这个线程异常结束，会有另一个取代它，保证顺序执行。单工作线程最大的特点是可保证顺序地执行各个任务，并且在任意给定的时间不会有多个线程是活动的。

**2.4 ScheduleThreadPool**

创建一个定长的线程池，而且支持定时的以及周期性的任务执行，支持定时及周期性任务执行。

**线程池任务的提交流程**

1. 如果当前线程池线程数目小于 corePoolSize（核心池还没满呢），那么就创建一个新线程去处理任务。
2. 如果核心池已经满了，来了一个新的任务后，会尝试将其添加到任务队列中，如果成功，则等待空闲线程将其从队列中取出并且执行，如果队列已经满了，则继续下一步。
3. 此时，如果线程池线程数量 小于 maximumPoolSize，则创建一个新线程执行任务，否则，那就说明线程池到了最大饱和能力了，没办法再处理了，此时就按照拒绝策略来处理。（就是构造函数当中的 Handler 对象）。
4. 如果线程池的线程数量大于 corePoolSize，则当某个线程的空闲时间超过了 keepAliveTime，那么这个线程就要被销毁了，直到线程池中线程数量不大于 corePoolSize 为止。

**线程池的 shutdown 和 shutdownNow 方法的区别是什么**

1. shutdown 设置状态为 SHUTDOWN，而 shutdownNow 设置状态为 STOP
2. shutdown 只中断空闲的线程，已提交的任务可以继续被执行，而 shutdownNow 中断所有线程
3. shutdown 无返回值，shutdownNow 返回任务队列中还未执行的任务

**线程池的参数要如何配置？（待整理给出）**

[http://www.cnblogs.com/waytobestcoder/p/5323130.html](https://link.jianshu.com?t=http:/www.cnblogs.com/waytobestcoder/p/5323130.html)

**单利模式DCL(双重检查)失效问题，如何写出一个高效的单利模式?**

单例模式，针对延迟加载法的同步实现所产生的性能低的问题，我们可以采用DCL，即双重检查加锁（Double Check Lock）的方法来避免每次调用getInstance（）方法时都同步。实现方式如下：

public class LazySingleton {

private int someField;

private static LazySingleton instance;

private LazySingleton() {

this.someField = new Random().nextInt(200)+1; // (1)

}

public static LazySingleton getInstance() {

if (instance == null) { // (2)

synchronized(LazySingleton.class) { // (3)

if (instance == null) { // (4)

instance = new LazySingleton(); // (5)

}

}

}

return instance; // (6)

}

public int getSomeField() {

return this.someField; // (7)

}

}

但是这种双重检查模式会在多线程的情况下出现线程不安全的情况，即调用 getInstance().getSomeField() 方法得到的值可能是其默认值，并非是初始化后的值。

**详细的原因参考笔记，篇幅的问题不展开**

**针对这种情况的解决方案**：

1. 最简单而且安全的解决方法是使用static内部类的思想，它利用的思想是：一个类直到被使用时才被初始化，而类初始化的过程是非并行的，这些都有JLS保证。

public class Singleton {

private Singleton() {}

private static class InstanceHolder {

private static final Singleton instance = new Singleton();

}

public static Singleton getSingleton() {

return InstanceHolder.instance;

}

}

1. 另外，可以将 instance 声明为 volatile。即
2. private volatile static LazySingleton instance;
3. 这样我们便可以得到，线程Ⅰ的语句(5) -> 语线程Ⅱ的句(2)，根据单线程规则，线程Ⅰ的语句(1) -> 线程Ⅰ的语句(5)和语线程Ⅱ的句(2) -> 语线程Ⅱ的句(7)，再根据传递规则就有线程Ⅰ的语句(1) -> 语线程Ⅱ的句(7)，这表示线程Ⅱ能够观察到线程Ⅰ在语句(1)时对someFiled的写入值，程序能够得到正确的行为。
4. 利用 java5 的 final 语义，final 变量一旦在构造函数中设置完成（前提是在构造函数中没有泄露this引用)，其它线程必定会看到在构造函数中设置的值。可以将 LazySingleton 的 someField 变量设置成 final，这样在 java5 中就能够正确运行了。

**线程同步与阻塞的关系?同步一定阻塞吗?阻塞一定同步吗?**

1. **同步和异步的关系**

同步和异步关注的是消息通信机制 (synchronous communication/ asynchronous communication)所谓同步，就是在发出一个**调用**时，在没有得到结果之前，该**调用**就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由**调用者**主动等待这个**调用**的结果。而异步则是相反，**调用**在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在**调用**发出后，**被调用者**通过状态、通知来通知调用者，或通过回调函数处理这个调用。

1. **阻塞与非阻塞的关系**

阻塞和非阻塞关注的是程序在**等待调用结果（消息，返回值）时的状态。**

**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。**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。

1. **总结**

同步是个过程，阻塞是线程的一种状态。多个线程操作共享变量时可能会出现竞争。这时需要同步来防止两个以上的线程同时进入临界区，在这个过程中，后进入临界区的线程将阻塞，等待先进入的线程走出临界区。

**同步不一定发生阻塞**，线程同步的时候，需要协调推进速度，相互等待和相互唤醒，因此会发生阻塞，等待另一个事件发生。线程阻塞原因很多，等待输入输出或者其他事件发生，不一定是因为同步产生的。**阻塞也不一定同步**，可能是线程等待着某个输入或者输出导致的阻塞。

**同步和异步有什么区别?**

同步和异步关注的是消息通信机制 (synchronous communication/ asynchronous communication)所谓同步，就是在发出一个**调用**时，在没有得到结果之前，该**调用**就不返回。但是一旦调用返回，就得到返回值了。换句话说，就是由**调用者**主动等待这个**调用**的结果。而异步则是相反，**调用**在发出之后，这个调用就直接返回了，所以没有返回结果。换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果。而是在**调用**发出后，**被调用者**通过状态、通知来通知调用者，或通过回调函数处理这个调用。

**Thread类中的yield方法有什么作用?**

yield 方法可以暂停当前正在执行的线程对象，让其它有相同优先级的线程执行。它是一个静态方法而且只保证当前线程放弃 CPU 占用而不能保证使其它线程一定能占用 CPU，执行 yield() 的线程有可能在进入到暂停状态后马上又被执行。

**产生死锁的四个必要条件，如何避免死锁**

1. 互斥条件：一个资源每次只能被一个进程使用。
2. 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
3. 不剥夺条件：进程已获得的资源，在末使用完之前，不能强行剥夺。
4. 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

避免死锁只要破坏其中的一个条件就可以了，其中最简单的方法就是阻止循环等待条件。

**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。在开发中如果想要处理这种未捕获的异常，可以实现这个接口，并在 ThreadFactory 中设置异常处理的 handler。

**ThreadLocal 的原理**

[http://www.cnblogs.com/lqminn/p/3751206.html](https://link.jianshu.com?t=http:/www.cnblogs.com/lqminn/p/3751206.html)

简单说 ThreadLocal 就是一种以空间换时间的做法，在每个Thread里面维护了一个 ThreadLocal.ThreadLocalMap，它以ThreadLocal为键，以属于该线程的资源副本为值。==我们可以这样看待ThreadLocal：ThreadLocal是为一组线程维护资源副本的对象，通过它，可以为每一个线程创建资源副本，也可以正确获得属于某一线程的资源副本。==

**应用场景**：当很多线程需要多次使用同一个对象，并且需要该对象具有相同初始化值的时候最适合使用ThreadLocal。

**ThreadLocal 是否会造成内存泄漏?** 答案是不会：每一个线程对资源副本都有一个隐式引用，当一个线程运行结束销毁时，所有的资源副本都是可以被垃圾回收的。也就是说当线程被回收的时候，那么 ThreadLocal 变量也会被回收。

**你如何在Java中获取线程堆栈?如何分析线程堆栈?（要精确到命令，重写）**

1. **如何获取**

不同的操作系统有不同的方式，在Windows你可以使用 Ctrl + Break 组合键来获取线程堆栈，Linux 下用 kill -3 命令。你也可以用 jstack 这个工具来获取，它对线程 id 进行操作，你可以用 jps 这个工具找到 id。

1. **如何分析**

**Java中synchronized 和 ReentrantLock 有什么不同？**

synchronized 是关键字， ReentrantLock 是类，这是它们本质的不同。 和synchronized相比，ReentrantLock用起来会复杂一些。在基本的加锁和解锁上，两者是一样的，所以无特殊情况下，推荐使用synchronized。ReentrantLock的优势在于它更灵活、更强大，增加了轮训、超时、中断等高级功能。

**怎么检测一个线程是否持有对象监视器**

Thread 类提供了一个 holdsLock(Object obj) 方法，当且仅当对象 obj 的监视器被某条线程持有的时候才会返回 true，注意这是一个 **static 方法**，这意味着"某条线程"指的是当前线程。

**CyclicBarrier 和 CountDownLatch 的区别**

**CyclicBarrier 主要用于一组线程之间的相互等待，而 CountDownLatch 一般用于一组线程等待另一组些线程。**

1. **CountDownLatch**

一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。

用给定的计数初始化 CountDownLatch。由于调用了 countDown() 方法，所以在当前计数到达 0 之前，await 方法会一直受阻塞。

之后，会释放所有等待的线程，await 的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。

1. **CyclicBarrier**
   * 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就象它名字的意思一样，可看成是个障碍， 所有的线程必须到齐后才能一起通过这个障碍。
   * 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

**CyclicBarrier 的实现原理**

CyclicBarrier 的字面意思是可循环（Cyclic）使用的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。线程进入屏障通过CyclicBarrier的await()方法。

CyclicBarrier 默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用 await 方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

CyclicBarrier 还提供一个更高级的构造函数 CyclicBarrier(int parties, Runnable barrierAction)，用于在线程到达屏障时，优先执行 barrierAction 这个 Runnable 对象，方便处理更复杂的业务场景。

**实现原理**：在 CyclicBarrier 的内部定义了一个 Lock 对象，每当一个线程调用 CyclicBarrier 的 await() 方法时，将剩余拦截的线程数减 1，然后判断剩余拦截数是否为 0，如果不是，进入 Lock 对象的条件队列等待。如果是，执行 barrierAction 对象的 Runnable 方法，然后将锁的条件队列中的所有线程放入锁等待队列中，这些线程会依次的获取锁、释放锁，接着先从 await() 方法返回，再从 CyclicBarrier 的 await 方法中返回。当最后一个线程到达屏障点，也就是执行 dowait 方法时，会在 return 0 返回之前调用 finally 块中的 breakBarrier 方法。

==CycliBarrier 对象可以重复使用，重用之前应当调用 CyclicBarrier 对象的 reset 方法。==

**CountDownLatch 实现原理**

**java 中的阻塞队列详细说明**

[http://ifeve.com/java-blocking-queue/](https://link.jianshu.com?t=http:/ifeve.com/java-blocking-queue/)

**什么是阻塞队列**

阻塞队列（BlockingQueue）是一个支持两个附加操作的队列。这两个附加的操作是：在队列为空时，获取元素的线程会等待队列变为非空。当队列满时，存储元素的线程会等待队列可用。阻塞队列常用于生产者和消费者的场景，生产者是往队列里添加元素的线程，消费者是从队列里拿元素的线程。阻塞队列就是生产者存放元素的容器，而消费者也只从容器里拿元素。

BlockingQueue 具有 4 组不同的方法用于插入、移除以及对队列中的元素进行检查。如果请求的操作不能得到立即执行的话，每个方法的表现也不同。这些方法如下：



image

**四组不同的行为方式解释**：

* 抛异常：如果试图的操作无法立即执行，抛一个异常。
* 特定值：如果试图的操作无法立即执行，返回一个特定的值(常常是 true / false)。
* 阻塞：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行。
* 超时：如果试图的操作无法立即执行，该方法调用将会发生阻塞，直到能够执行，但等待时间不会超过给定值。返回一个特定值以告知该操作是否成功(典型的是true / false)。

无法向一个 BlockingQueue 中插入 null。如果你试图插入 null，BlockingQueue 将会抛出一个 NullPointerException。

**JDK7提供了7个阻塞队列，分别是**：

* ArrayBlockingQueue ：一个由数组结构组成的有界阻塞队列。
* LinkedBlockingQueue ：一个由链表结构组成的有界阻塞队列。
* PriorityBlockingQueue ：一个支持优先级排序的无界阻塞队列。
* DelayQueue：一个使用优先级队列实现的无界阻塞队列。
* SynchronousQueue：一个不存储元素的阻塞队列。
* LinkedTransferQueue：一个由链表结构组成的无界阻塞队列。
* LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列。

BlockingQueue 是个接口，你需要使用它的实现之一来使用 BlockingQueue，java.util.concurrent 包下具有以下 BlockingQueue 接口的实现类：

* **ArrayBlockingQueue**：ArrayBlockingQueue 是一个有界的阻塞队列，其内部实现是将对象放到一个数组里。有界也就意味着，它不能够存储无限多数量的元素。它有一个同一时间能够存储元素数量的上限。你可以在对其初始化的时候设定这个上限，但之后就无法对这个上限进行修改了(译者注：因为它是基于数组实现的，也就具有数组的特性：一旦初始化，大小就无法修改)。
* **DelayQueue**：DelayQueue 对元素进行持有直到一个特定的延迟到期。注入其中的元素必须实现 java.util.concurrent.Delayed 接口。
* **LinkedBlockingQueue**：LinkedBlockingQueue 内部以一个链式结构(链接节点)对其元素进行存储。如果需要的话，这一链式结构可以选择一个上限。如果没有定义上限，将使用 Integer.MAX\_VALUE 作为上限。
* **PriorityBlockingQueue**：PriorityBlockingQueue 是一个无界的并发队列。它使用了和类 java.util.PriorityQueue 一样的排序规则。你无法向这个队列中插入 null 值。所有插入到 PriorityBlockingQueue 的元素必须实现 java.lang.Comparable 接口。因此该队列中元素的排序就取决于你自己的 Comparable 实现。
* **SynchronousQueue**：SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素。如果该队列已有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从队列中抽走。同样，如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另一个线程向队列中插入了一条新的元素。据此，把这个类称作一个队列显然是夸大其词了。它更多像是一个汇合点。

**阻塞队列原理**：其实阻塞队列实现阻塞同步的方式很简单，使用的就是是lock锁的多条件（condition）阻塞控制。使用BlockingQueue封装了根据条件阻塞线程的过程，而我们就不用关心繁琐的await/signal操作了。

**ReadWriteLock 是什么**

ReadWriteLock 是一个读写锁接口，ReentrantReadWriteLock 是 ReadWriteLock 接口的一个具体实现，实现了读写的分离，读锁是共享的，写锁是独占的，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

**Linux 环境下如何查找哪个线程使用 CPU 最长**

1. 获取项目的pid，jps或者ps -ef | grep java
2. top -H -p pid，顺序不能改变

这样就可以打印出当前的项目，每条线程占用CPU时间的百分比。注意这里打出的是LWP，也就是操作系统原生线程的线程号。打出来的LWP是十进制的，"jps pid"打出来的本地线程号是十六进制的，转换一下，就能定位到占用CPU高的线程的当前线程堆栈了。

使用"top -H -p pid"+"jps pid"可以很容易地找到某条占用CPU高的线程的线程堆栈，从而定位占用CPU高的原因，一般是因为不当的代码操作导致了死循环。

**Thread.sleep(0) 的作用是什么（要弄懂）**

由于Java采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到CPU控制权的情况，为了让某些优先级比较低的线程也能获取到CPU控制权，可以使用Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡CPU控制权的一种操作。

**Java 内存模型**

**概括总结**：

JMM 是 Java 程序对线程如何交互的统一的约定协议。Java 内存模型是围绕着并发编程中**原子性、可见性、有序性**这三个特征来建立的。**原子性**：一个操作是最小单元，不可分割。**可见性**：一个线程的修改对另一个线程可见。

在 JMM 中规定了 Happens-Before 顺序: 保证了一个线程的操作结果能够对另一个线程可见。

synchronized 关键字提供互斥区和内存可见性, 防止重排序。

volatile 提供内存可见性,防止重排序，保证 64 位元素(double、long)的原子性读写。

对 final 语义的增强，使得被 final 修饰的变量或者引用在特定的情况下不能被重排序。

[https://liuzhengyang.github.io/2017/05/12/javamemorymodel/](https://link.jianshu.com?t=https:/liuzhengyang.github.io/2017/05/12/javamemorymodel/)

Java Memory Model 简称 JMM, 是一系列的 Java 虚拟机平台对开发者提供的多线程环境下的内存可见性、是否可以重排序等问题的无关具体平台的统一的保证。Java 内存模型是围绕着并发编程中原子性、可见性、有序性这三个特征来建立的。

**插入内存屏障**

单处理器上由于会保证透明的顺序一致性，所以并不需要明确的插入内存屏障。

在多处理器情况下，基于上面的规则，可以在 volatile 字段、synchronized 关键字的处理上增加屏障来满足内存模型的规则。

最保守的策略是在volatile的前后都加上所有的屏障，但是这样在大多数情况都是不必要且重复的，所以再以volatile字段通常读多写少的假设，可以得出以下一种策略供编译器开发者参考:

1. volatile store前插入LoadStore;StoreStore屏障
2. 所有final字段写入后但在构造器返回前插入StoreStore
3. volatile store后插入StoreLoad屏障
4. 在volatile load后插入LoadLoad和LoadStore屏障
5. monitor enter和volatile load规则一致，monitor exit 和volatile store规则一致。

**内存屏障的规则**

| **需要的屏障** | **第二个操作** | **第二个操作** | **第二个操作** | **第二个操作** |
| --- | --- | --- | --- | --- |
| 第一个操作 | 普通读 | 普通写 | volatile读/monitor enter | volatile写/monitor exit |
| 普通读 |  |  |  | LoadStore |
| 普通读 |  |  |  | StoreStore |
| voaltile读/monitor enter | LoadLoad | LoadStore | LoadLoad | LoadStore |
| volatile写/monitor exit |  |  | StoreLoad | StoreStore |

**什么是 Java 内存模型(重新写)**

[http://www.infoq.com/cn/articles/java-memory-model-1](https://link.jianshu.com?t=http:/www.infoq.com/cn/articles/java-memory-model-1)

[http://www.cnblogs.com/skywang12345/p/3447546.html](https://link.jianshu.com?t=http:/www.cnblogs.com/skywang12345/p/3447546.html)

[http://ifeve.com/java-memory-model-6/](https://link.jianshu.com?t=http:/ifeve.com/java-memory-model-6/)

**什么是乐观锁和悲观锁**

**乐观锁**：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

**悲观锁**：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像 synchronized，不管三七二十一，直接上了锁就操作资源了。

**Semaphore 有什么作用**

Semaphore 就是一个信号量，它的作用是限制某段代码块的并发数。Semaphore 有一个构造函数，可以传入一个 int 型整数 n，表示某段代码最多只有 n 个线程可以访问，如果超出了 n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果 Semaphore 构造函数中传入的 int 型整数 n = 1，相当于变成了一个 synchronized 了。

**使用场景**：Semaphore可以用于做流量控制，特别公用资源有限的应用场景，比如数据库连接。

**高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？**

1. 高并发、任务执行时间短的业务，线程池线程数可以设置为 CPU 核数 +1，减少线程上下文的切换
2. 并发不高、任务执行时间长的业务要区分开看：
   * 假如是业务时间长集中在 IO 操作上，也就是 IO 密集型的任务，因为 IO 操作并不占用 CPU，所以不要让所有的 CPU 闲下来，可以加大线程池中的线程数目，让 CPU 处理更多的业务
   * 假如是业务时间长集中在计算操作上，也就是计算密集型任务，只能把线程池中的线程数设置得少一些，减少线程上下文的切换
3. 并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

**Fork/Join 框架的理解**

[http://www.infoq.com/cn/articles/fork-join-introduction](https://link.jianshu.com?t=http:/www.infoq.com/cn/articles/fork-join-introduction)

Fork/Join 框架是 Java7 提供了的一个用于并行执行任务的框架， 是一个把大任务分割成若干个小任务，最终汇总每个小任务结果后得到大任务结果的框架。Fork 就是把一个大任务切分为若干子任务并行的执行，Join 就是合并这些子任务的执行结果，最后得到这个大任务的结果。Fork/Join 的核心是 work-stealing 算法。

Fork/Join 使用两个类来完成以上两件事情：

* **ForkJoinTask**：我们要使用 ForkJoin 框架，必须首先创建一个 ForkJoin 任务。它提供在任务中执行 fork() 和 join() 操作的机制，通常情况下我们不需要直接继承 ForkJoinTask 类，而只需要继承它的子类，Fork/Join 框架提供了以下两个子类：
  + RecursiveAction：用于没有返回结果的任务。
  + RecursiveTask ：用于有返回结果的任务。
* **ForkJoinPool** ：ForkJoinTask 需要通过 ForkJoinPool 来执行，任务分割出的子任务会添加到当前工作线程所维护的双端队列中，进入队列的头部。当一个工作线程的队列里暂时没有任务时，它会随机从其他工作线程的队列的尾部获取一个任务。

**ForkJoinTask 与一般的任务的主要区别在于**它需要实现 compute 方法，在这个方法里，首先需要判断任务是否足够小，如果足够小就直接执行任务。如果不足够小，就必须分割成两个子任务，每个子任务在调用 fork 方法时，又会进入 compute 方法，看看当前子任务是否需要继续分割成孙任务，如果不需要继续分割，则执行当前子任务并返回结果。使用 join 方法会等待子任务执行完并得到其结果。

ForkJoinTask 在执行的时候可能会抛出异常，但是我们没办法在主线程里直接捕获异常，所以 ForkJoinTask 提供了 isCompletedAbnormally() 方法来检查任务是否已经抛出异常或已经被取消了，并且可以通过 ForkJoinTask 的 getException 方法获取异常。

**work-stealing 工作窃取算法**

所谓 Work-Stealing，在 ForkJoinPool 中的实现为：线程池中每个线程都有一个互不影响的任务队列（双端队列），线程每次都从自己的任务队列的队头中取出一个任务来运行；如果某个线程对应的队列已空并且处于空闲状态，而其他线程的队列中还有任务需要处理但是该线程处于工作状态，那么空闲的线程可以从其他线程的队列的队尾取一个任务来帮忙运行 —— 感觉就像是空闲的线程去偷人家的任务来运行一样，所以叫 “工作窃取”。

Work-Stealing 的适用场景是不同的任务的耗时相差比较大，即某些任务需要运行较长时间，而某些任务会很快的运行完成，这种情况下用 Work-Stealing 很合适；但是如果任务的耗时很平均，则此时 Work-Stealing 并不适合，因为窃取任务时不同线程需要抢占锁，这可能会造成额外的时间消耗，而且每个线程维护双端队列也会造成更大的内存消耗。所以 ForkJoinPool 并不是 ThreadPoolExecutor 的替代品，而是作为对 ThreadPoolExecutor 的补充。

**ForkJoinPool 和 ThreadPoolExecutor 的区别**

ForkJoinPool 和 ThreadPoolExecutor 都是 ExecutorService（线程池），但ForkJoinPool 的独特点在于：

1. ThreadPoolExecutor 只能执行 Runnable 和 Callable 任务，而 ForkJoinPool 不仅可以执行 Runnable 和 Callable 任务，还可以执行 Fork/Join 型任务 —— ForkJoinTask —— 从而满足并行地实现分治算法的需要。
2. ThreadPoolExecutor 中任务的执行顺序是按照其在共享队列中的顺序来执行的，所以后面的任务需要等待前面任务执行完毕后才能执行，而 ForkJoinPool 每个线程有自己的任务队列，并在此基础上实现了 Work-Stealing 的功能，使得在某些情况下 ForkJoinPool 能更大程度的提高并发效率。

**forkjoin 框架和 mapreduce 框架有什么区别?**

MapReduce 是把大数据集切分成小数据集，并行分布计算后再合并。

ForkJoin 是将一个问题递归分解成子问题，再将子问题并行运算后合并结果。

**二者共同点**：都是用于执行并行任务的。基本思想都是把问题分解为一个个子问题分别计算，再合并结果。应该说并行计算都是这种思想，彼此独立的或可分解的。从名字上看 Fork 和 Map 都有切分的意思，Join 和 Reduce 都有合并的意思，比较类似。

**区别**：

1. 环境差异，分布式 vs 单机多核：ForkJoin 设计初衷针对单机多核（处理器数量很多的情况）。MapReduce 一开始就明确是针对很多机器组成的集群环境的。也就是说一个是想充分利用多处理器，而另一个是想充分利用很多机器做分布式计算。这是两种不同的的应用场景，有很多差异，因此在细的编程模式方面有很多不同。
2. 编程差异：MapReduce 一般是：做较大粒度的切分，一开始就先切分好任务然后再执行，并且彼此间在最后合并之前不需要通信。这样可伸缩性更好，适合解决巨大的问题，但限制也更多。ForkJoin 可以是较小粒度的切分，任务自己知道该如何切分自己，递归地切分到一组合适大小的子任务来执行，因为是一个 JVM 内，所以彼此间通信是很容易的，更像是传统编程方式。

**二、集合**

**ConcurrentHashMap JDK 1.7 的实现原理**

ConcurrentHashMap 在 1.7 中采用了相同的数据结构，即分段锁的技术来实现的。ConcurrentHashMap 内部有一个叫 Segment 的数组，里面存放的都是 Segment 对象。Segment 对象继承了 ReentrantLock，这样就使得每个段都有一把锁。 Segment 里面有一个被 volatile 修饰的 HashEntry 的数组。（在 ConcurrentHashMap 初始化的时候，创建了 Segment 数组，并初始化第一个元素。）

**put 方法**

1. 判断 value 是否为 null， 如果 value 为 null 则抛出异常。
2. 当用户调用 put 方法的时候，首先根据 key 的 hash 值找到具体的 Segment 在 table 中的位置。
3. 如果这个位置上的 Segment 没有初始化，则进行初始化的操作。
4. 最后委托给 Segment 的 put 方法。此方法中会根据计算出的 (tab.length - 1) & hash 的 index 定位到 HashEntry， 如果这个位置上的节点为null，则新建一个 HashEntry并返回 。如果不为 null，则遍历 HashEntry 的每一个节点，如果有相同的 key 存在则更新 value， 如果没有则新建一个 HashEntry 放入**链表头部**的位置。
5. 如果 ConcurrentHashMap 内存放的元素个数超过了阈值，那么需要对其进行扩容。整个操作都是加锁的。

**get 操作**

==get 操作的时候没有对 ConcurrentHashMap 进行上锁==

1. 根据 key 的 hash 值计算出在哪个 Segment 上，再根据 hash 值计算出在哪个 HashEntry 上
2. 然后遍历 HashEntry 的所有节点，如果找到 key，那么就返回对应的 value，如果 key 没有找到，就返回 null。

**扩容**

ConcurrentHashMap不会增加Segment的数量，而只会增加Segment中链表数组的容量大小，这样的好处是扩容过程不需要对整个ConcurrentHashMap做rehash，而只需要对Segment里面的元素做一次 resize 就可以了。

**ConcurrentHashMap JDK 1.8 的实现原理**

ConcurrentHashMap 在 JDK 1.8 中进行了大幅度的改进。取消了 Segment 分段锁的概念。采用了数组 + 链表 + 红黑树的数据结构实现。内部存放了一个 Node<K,V>[] table 的 table。 ConcurrentHashMap 在初始化的时候只是设置了一些变量值，并没有对整个 table 进行初始化，初始化的动作被放入到了第一次 put 元素的时候。当链表长度大于 8 并且数组的长度大于 64 的时候会将链表转成红黑树结构。

**put 操作**

1. 首先判断 key 和 value 是否为 null， 如果为 null 则抛出异常。
2. 然后在判断 table 是否初始化，如果没有初始化则通过 CAS 操作将 sizeCtl 的值设置为 -1，并执行 table 的初始化操作。
3. 根据 key 的 hash 定位到 key 所在 table 的位置，如果这个位置上没有元素，则直接插入元素后返回。
4. 如果当前节点的 hash 值为 -1，说明当前的节点是 forwardingNode 节点，表示 table 正在扩容，当前线程需要帮助一起扩容。（上面的过程走完之后，说明当前的节点上有元素，需要对当前节点加锁然后操作）。
5. 如果当前节点的 hash 值大于等于 0，说明是一个链表结构，则遍历链表，如果存在当前 key 节点则替换 value，否则插入到链表尾部。
6. 如果 f 是 TreeBin 类型节点，则按照红黑树的方法更新或者增加节点。
7. 若链表长度 > TREEIFY\_THRESHOLD(默认是8)，则将链表转换为红黑树结构（并不是直接转的，还需要进一步判断，具体的在treeifyBin()方法中）。
8. 最后调用 addCount 方法，将 ConcurrentHashMap 的 size + 1，并判断是否需要执行扩容操作，整个 put 过程结束。

**get 操作**

get 操作的时候没有上锁，如果整个table 为空，则返回null，否则根据 key 的 hash 值找到 table 的 index 位置，然后根据链表或者树形方式找到相对应的节点，返回其 value 值。

**红黑树转换**

链表的元素个数达到了阈值 8 ，则会调用 treeifyBin 方法把链表转换成红黑树，不过在结构转换之前，会对数组长度进行判断。如果数组长度 n 小于阈值 MIN\_TREEIFY\_CAPACITY，默认是 64，则会调用 tryPresize 方法把数组长度扩大到原来的两倍，并触发 transfer 方法，重新调整节点的位置。

**扩容操作**

整个扩容操作分为两步：

1. 构建一个nextTable，其大小为原来大小的两倍，这个步骤是在单线程环境下完成的。
2. 将原来table里面的内容复制到nextTable中，这个步骤是允许多线程操作的，所以性能得到提升，减少了扩容的时间消耗。

并发扩容的具体步骤如下：

1. 为每个内核分任务，并保证其不小于16
2. 检查nextTable是否为null，如果是，则初始化 nextTable，使其容量为 table 的两倍。然后死循环遍历节点，直到finished。节点从 table 复制到 nextTable 中，支持并发，思路如下：
3. 如果节点 f 为 null，则插入 ForwardingNode（采用 Unsafe.compareAndSwapObject 方法实现），这个是触发并发扩容的关键
4. 如果 f 为链表的头节点（fh >= 0）,则先构造一个**反序链表**，然后把他们分别放在nextTable的 i 和 i + n位置，并将 ForwardingNode 插入原节点位置，代表已经处理过了
5. 如果 f 为 TreeBin 节点，同样也是构造一个**反序链表** ，==同时需要判断是否需要进行 unTreeify() 操作==，并把处理的结果分别插入到 nextTable 的 i 和 i+n 位置，并插入 ForwardingNode 节点
6. 所有节点复制完成后，则将 table 指向 nextTable，同时更新 sizeCtl = nextTable 的 0.75 倍，完成扩容过程。

在多线程环境下，ConcurrentHashMap 用两点来保证正确性：ForwardingNode 和 synchronized。当一个线程遍历到的节点如果是 ForwardingNode，则继续往后遍历，如果不是，则将该节点加锁，防止其他线程进入，完成后设置 ForwardingNode 节点，以便要其他线程可以看到该节点已经处理过了，如此交叉进行，高效而又安全。

**HashMap JDK 1.7 实现原理**

HashMap 在 JDK 7 中的数据结构是数组 + 链表的实现。 HashMap 中存储了一个 Entry[] 类型的数组，里面存储了 Entry 对象。

**put**

1. 当调用 put 方法的时候，会根据 key 的 hash 值定位到 key 要存到 table 的哪个 index 上
2. 如果 key 为 null，那么就 put 到链表的头结点上。
3. 如果 key 不为 null，那么遍历 index 上的链表，如果存在相同的 key，那么就更新 value。
4. 如果不存在相同的 key，那么将 key 存到链表的头结点中。

**get**

1. 根据 key 的 hash 值计算出 key 在 table 的哪个 index 上。遍历 index 上的链表，找到相同的 key，并返回 value。
2. 如果 key 不存在则返回 null。

**扩容**

**什么时候扩容?**

当 HashMap 内的容量数超过了阈值（默认 12 个）的时候会触发扩容。整个扩容过程如下：

1. 新建一个比原来数组两倍大的新数组。
2. 重算 key 的 hash 值来得到在新数组的位置，并将 key 放入新数组中。

**死循环问题**

主要是多线程同时put时，如果同时触发了rehash操作，会导致HashMap中的链表中出现循环节点，进而使得后面get的时候，会死循环。而且还会丢失元素。

**HashMap JDK 1.8 实现原理**

在 JDK 1.8 中， HashMap 重新设计了实现。放弃 1.7 中的数组 + 链表的存储结构，改为了数组 + 链表 + 红黑树的实现。

**put**

1. 根据 key 的 hash 值，计算出 table 中的 index。
2. 如果 index 上没有元素，那么直接插入元素。
3. 如果 index 上有元素的话，并且是链表结构的话，就遍历链表，判断是否有相同的 key 存在，如果存在则替换 value，如果不存在则新建 Node **==放入链表尾部==**。同时判断当前链表是否过长，如果超过 TREEIFY\_THRESHOLD（默认 8 个） 的话，则需要将链表转换成红黑树。
4. 如果 index 上的节点是 TreeNode 类型的话，则用红黑树的方式添加元素。
5. 最后判断 HashMap 中的元素是否超过了阈值，如果超过了需要进行 resize 扩容。

**get**

1. 根据 key 的 hash 值定位到 table 中的 index。
2. 如果 index 上没有元素，则返回 null。
3. 如果 index 上有元素，那么根据节点类型的不同，调用链表或红黑树的方式获取 value。

**扩容**

在 JDK 1.8 的实现中，优化了高位运算的算法，通过 hashCode() 的高 16 位异或低 16 位实现的：(h = k.hashCode()) ^ (h >>> 16)，主要是从速度、功效、质量来考虑的，这么做可以在数组table的length比较小的时候，也能保证考虑到高低Bit都参与到Hash的计算中，同时不会有太大的开销。

**ArrayList 和 LinkedList 的区别**

1．对ArrayList和LinkedList而言，在列表末尾增加一个元素所花的开销都是固定的。对ArrayList而言，主要是在内部数组中增加一项，指向所添加的元素，偶尔可能会导致对数组重新进行分配；而对LinkedList而言，这个开销是统一的，分配一个内部Entry对象。

2．在ArrayList的中间插入或删除一个元素意味着这个列表中剩余的元素都会被移动；而在LinkedList的中间插入或删除一个元素的开销是固定的。

3．LinkedList不支持高效的随机元素访问。

4．ArrayList的空间浪费主要体现在在list列表的结尾预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗相当的空间

可以这样说：当操作是在一列数据的后面添加数据而不是在前面或中间,并且需要随机地访问其中的元素时,使用ArrayList会提供比较好的性能；当你的操作是在一列数据的前面或中间添加或删除数据,并且按照顺序访问其中的元素时,就应该使用LinkedList了。

**三、IO/NIO**

**TCP协议与UDP协议的区别**

1. TCP 是面向连接的，而 UDP 是面向无连接的
2. 对系统资源的要求（TCP 较多，UDP 少）
3. UDP 程序结构较简单
4. TCP 采用流模式，而 UDP 采用数据报文模式
5. TCP 保证数据正确性，UDP 可能丢包，TCP 保证数据顺序，UDP 不保证。

**三次握手和四次挥手的过程**

**三次握手**

所谓三次握手(Three-way Handshake)，是指建立一个 TCP 连接时，需要客户端和服务器总共发送 3 个包。

三次握手的目的是连接服务器指定端口，建立 TCP 连接,并同步连接双方的序列号和确认号并交换 TCP 窗口大小信息.在 socket 编程中，客户端执行 connect() 时。将触发三次握手。

**第一次握手**: 客户端发送一个 TCP 的 SYN 标志位置1的包指明客户打算连接的服务器的端口，以及初始序号 X,保存在包头的序列号(Sequence Number)字段里。

**第二次握手**：服务器发回确认包(ACK)应答。即 SYN 标志位和 ACK 标志位均为 1 同时，将确认序号(Acknowledgement Number)设置为客户的 ISN 加 1 以.即 X + 1。

**第三次握手**：客户端再次发送确认包(ACK) SYN 标志位为 0,ACK 标志位为1。并且把服务器发来 ACK 的序号字段 +1，放在确定字段中发送给对方。并且在数据段放写 ISN 的 +1。

**四次挥手**

客户端或服务器均可主动发起挥手动作，在socket编程中，任何一方执行close()操作即可产生挥手操作。

**第一次分手**：主机1（可以使客户端，也可以是服务器端），设置Sequence Number和Acknowledgment Number，向主机2发送一个FIN报文段；此时，主机1进入FIN\_WAIT\_1状态；这表示主机1没有数据要发送给主机2了；

**第二次分手**：主机 2 收到了主机 1 发送的 FIN 报文段，向主机 1 回一个 ACK 报文段，Acknowledgment Number 为 Sequence Number 加 1；主机 1 进入 FIN\_WAIT\_2 状态；主机 2 告诉主机 1，我“同意”你的关闭请求；

**第三次分手**：主机 2 向主机 1 发送 FIN 报文段，请求关闭连接，同时主机 2 进入 LAST\_ACK 状态；

**第四次分手**：主机 1 收到主机 2 发送的 FIN 报文段，向主机 2 发送 ACK 报文段，然后主机 1 进入 TIME\_WAIT 状态；主机 2 收到主机 1 的 ACK 报文段以后，就关闭连接；此时，主机 1 等待 2MS 后依然没有收到回复，则证明 Server 端已正常关闭，那好，主机 1 也可以关闭连接了。

**java 中的字节顺序**

**字节顺序是指**：占用内存多于一个字节类型的数据在内存中的存放顺序，有小端、大端两种顺序。**小端字节序**（little endian）：低字节数据存放在内存低地址处，高字节数据存放在内存高地址处；**大端字节序**（bigendian）：高字节数据存放在低地址处，低字节数据存放在高地址处。

至于计算机到底是 BIG-ENDIAN、LITTLE-ENDIAN、跟 CPU 有关的，一种 CPU 不是 BIG-ENDIAN 就是 LITTLE-ENDIAN。IA 架构(Intel、AMD)的 CPU 中是 Little-Endian，而 PowerPC 、SPARC 和 Motorola 处理器是 Big-Endian。这其实就是所谓的主机字节序。而网络字节序是指数据在网络上传输时是大头还是小头的，在 Internet 的网络字节序是 BIG-ENDIAN。所谓的 JAVA 字节序指的是在 JAVA 虚拟机中多字节类型数据的存放顺序，JAVA 字节序也是 BIG-ENDIAN。

JDK为我们提供一个类ByteOrder，通过以下代码就可以知道机器的字节序

System.out.println(ByteOrder.nativeOrder());

在java.nio包下提供了ByteOrder、ByteBuffer等于字节序相关的类，我们也可以改变JVM中默认的字节序。代码如下：

public class JVMEndianTest {

public static void main(String[] args) {

int x = 0x01020304;

ByteBuffer bb = ByteBuffer.wrap(new byte[4]);

bb.asIntBuffer().put(x);

String ss\_before = Arrays.toString(bb.array());

System.out.println("默认字节序 " + bb.order().toString() + "," + " 内存数据 " + ss\_before);

bb.order(ByteOrder.LITTLE\_ENDIAN);

bb.asIntBuffer().put(x);

String ss\_after = Arrays.toString(bb.array());

System.out.println("修改字节序 " + bb.order().toString() + "," + " 内存数据 " + ss\_after);

}

}

执行结果如下：

默认字节序 BIG\_ENDIAN, 内存数据 [1, 2, 3, 4]

修改字节序 LITTLE\_ENDIAN, 内存数据 [4, 3, 2, 1]

**socket 的一些重要参数**

[http://www.cnblogs.com/ggjucheng/archive/2012/01/06/2314679.html](https://link.jianshu.com?t=http:/www.cnblogs.com/ggjucheng/archive/2012/01/06/2314679.html)

1. **backlog**

连接请求的最大队列长度被设置为 backlog 参数。如果队列满时则拒绝该连接。

* + backlog 这个参数设置为 -1 表示无限制，默认是 50 个最大等待队列
  + 经过测试这个队列是按照 FIFO（先进先出）的原则。
  + 如果将 accept 这个函数放在一个循环体中时，backlog 参数也不会有什么作用。

1. **TcpNoDelay**

禁用纳格算法，将数据立即发送出去。纳格算法是以减少封包传送量来增进TCP/IP网络的效能。

1. **SoLinger**

当我们调用 socket.close() 返回时，socket 已经 write 的数据未必已经发送到对方了，那么我们设置了 socket.setSoLinger(true, 100) 时，那么 close 会等到发送的数据已经确认了才返回。但是如果对方宕机，超时，那么会根据 linger 设定的时间返回。

1. **SoTimeout**

设置 socket 调用 InputStream 读数据的超时时间，以毫秒为单位，如果超过这个时候，会抛出 java.net.SocketTimeoutException。

1. **KeepAlive**

keepalive 不是说 TCP 的常连接，当我们作为服务端，一个客户端连接上来，如果设置了 keeplive 为 true，当对方没有发送任何数据过来，超过一个时间(看系统内核参数配置)，那么我们这边会发送一个 ack 探测包发到对方，探测双方的 TCP/IP 连接是否有效(对方可能断点，断网)，在 Linux 好像这个时间是 75 秒。如果不设置，那么客户端宕机时，服务器永远也不知道客户端宕机了，仍然保存这个失效的连接。

1. **SendBufferSize和ReceiveBufferSize**

TCP 发送缓存区和接收缓存区,默认是 8192，一般情况下足够了，而且就算你增加了发送缓存区，对方没有增加它对应的接收缓冲，那么在 TCP 三握手时，最后确定的最大发送窗口还是双方最小的那个缓冲区，就算你无视，发了更多的数据，那么多出来的数据也会被丢弃。除非双方都协商好。

**ByteBuffer 如何创建，如何读写?**

作者：Java旅行者  
链接：https://www.jianshu.com/p/ac6b889e73c5  
來源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。