**一：多线程基础题**

**1、 什么是线程？**

线程是操作系统能够进行运算调度的最小单位，它被包含在进程之中，是进程中的实际运作单位。

程序员可以通过它进行多处理器编程，你可以使用多线程对运算密集型任务提速。

比如，如果一个线程完成一个任务要100毫秒，那么用十个线程完成该任务只需10毫秒。

**2、线程和进程有什么区别？**

一个进程是一个独立(self contained)的运行环境，它可以被看作一个程序或者一个应用。而线程是在进程中执行的一个任务。

线程是进程的子集，一个进程可以有很多线程，每条线程并行执行不同的任务。

不同的进程使用不同的内存空间，而所有的线程共享一片相同的内存空间。

每个线程都拥有单独的栈内存用来存储本地数据。

**3、 如何在Java中实现线程？**

**（1）.继承Thread类实现多线程**

继承Thread类,然后重写run方法.（由于Java单继承的特性，这种方式用的比较少）

public class MyThread extends Thread {

public MyThread() {

}

public void run() {

for(int i=0;i<10;i++) {

System.out.println(Thread.currentThread()+":"+i);

}

}

public static void main(String[] args) {

MyThread mThread1=new MyThread();

MyThread mThread2=new MyThread();

MyThread myThread3=new MyThread();

mThread1.start();

mThread2.start();

myThread3.start();

}

}

**（2）.实现Runnable()接口定制执行目标（target）类，实现其run()方法**

推荐此方式。两个特点：

* a.覆写Runnable接口实现多线程可以避免单继承局限
* b.实现Runnable()可以更好的体现共享的概念
* c.当执行目标类实现Runnable接口，此时执行目标（target）类和Thread是代理模式（子类负责真是业务的操作，thread负责资源调度与线程创建辅助真实业务。

public class MyTarget implements Runnable{

public static int count=20;

public void run() {

while(count>0) {

try {

Thread.sleep(200);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(Thread.currentThread().getName()+"-当前剩余票数:"+count--);

}

}

public static void main(String[] args) {

MyThread target=new MyTarget();

Thread mThread1=new Thread(target,"线程1");

Thread mThread2=new Thread(target,"线程2");

Thread mThread3=new Thread(target,"线程3");

mThread1.start();

mThread2.start();

myThread3.start();

}

}

**（3）.实现Callable接口创建多线程（JDK1.5）**

​ a.执行目标核心方法叫call()方法  
​ b.有返回值

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutionException;

import java.util.concurrent.FutureTask;

public class MyTarget implements Callable<String> {

private int count = 20;

@Override

public String call() throws Exception {

for (int i = count; i > 0; i--) {

*// Thread.yield();*

System.out.println(Thread.currentThread().getName()+"当前票数：" + i);

}

return "sale out";

}

public static void main(String[] args) throws InterruptedException, ExecutionException {

Callable<String> callable =new MyTarget();

FutureTask <String>futureTask=new FutureTask<>(callable);

Thread mThread=new Thread(futureTask);

Thread mThread2=new Thread(futureTask);

Thread mThread3=new Thread(futureTask);

*// mThread.setName("hhh");*

mThread.start();

mThread2.start();

mThread3.start();

System.out.println(futureTask.get());

}

}

**（4）.通过线程池创建多线程**

请参见后面线程池的面试题。

**4、 继承Thread类或者实现Runnable接口来创建线程，有何区别？**

这个问题是上题的后续，大家都知道我们可以通过继承Thread类或者调用Runnable接口来实现线程，问题是，那个方法更好呢？什么情况下使用它？

提示：这个问题很容易回答。Java不支持类的多重继承，但允许你调用多个接口。所以如果你要继承其他类，当然是调用Runnable接口好了。

参考答案，有两点：

* a.覆写Runnable接口实现多线程可以避免单继承局限
* b.实现Runnable()可以更好的体现共享的概念

**5、 Thread 类中的start() 和 run() 方法有什么区别？**

start()方法被用来启动新创建的线程，使该被创建的线程状态变为可运行状态。

当你直接调用run()方法的时候，只会是在原来的线程中调用，没有新的线程启动。只有调用start()方法才会启动新线程。

如果我们调用了Thread的run()方法，它的行为就会和普通的方法一样，直接运行run（）方法。为了在新的线程中执行我们的代码，必须使用Thread.start()方法。

**6、Java中Runnable和Callable有什么不同？**

Runnable和Callable都代表那些要在不同的线程中执行的任务目标target。

Runnable从JDK1.0开始就有了，Callable是在JDK1.5增加的。它们的主要区别是Callable的 call() 方法可以返回值和抛出异常，而Runnable的run()方法没有这些功能。

**7、线程的常用方法有哪些**

1. Thread.sleep(long millis)，一定是当前线程调用此方法，当前线程进入TIMED\_WAITING状态，但不释放对象锁，millis后线程自动苏醒进入就绪状态。作用：给其它线程执行机会的最佳方式。
2. Thread.yield()，一定是当前线程调用此方法，当前线程放弃获取的CPU时间片，但不释放锁资源，由运行状态变为就绪状态，让OS再次选择线程。作用：让相同优先级的线程轮流执行，但并不保证一定会轮流执行。实际中无法保证yield()达到让步目的，因为让步的线程还有可能被线程调度程序再次选中。Thread.yield()不会导致阻塞。该方法与sleep()类似，只是不能由用户指定暂停多长时间。
3. thread.join()/thread.join(long millis)，当前线程里调用其它线程t的join方法，当前线程进入WAITING/TIMED\_WAITING状态，当前线程不会释放已经持有的对象锁。线程t执行完毕或者millis时间到，当前线程一般情况下进入RUNNABLE状态，也有可能进入BLOCKED状态（因为join是基于wait实现的）。
4. obj.wait()，当前线程调用对象的wait()方法，当前线程释放对象锁，进入等待队列。依靠notify()/notifyAll()唤醒或者wait(long timeout) timeout时间到自动唤醒。
5. obj.notify()唤醒在此对象监视器上等待的单个线程，选择是任意性的。notifyAll()唤醒在此对象监视器上等待的所有线程。
6. LockSupport.park()/LockSupport.parkNanos(long nanos),LockSupport.parkUntil(long deadlines), 当前线程进入WAITING/TIMED\_WAITING状态。对比wait方法,不需要获得锁就可以让线程进入WAITING/TIMED\_WAITING状态，需要通过LockSupport.unpark(Thread thread)唤醒。

**8) Java内存模型是什么？**

Java内存模型规定和指引Java程序在不同的内存架构、CPU和操作系统间有确定性地行为。它在多线程的情况下尤其重要。Java内存模型对一个线程所做的变动能被其它线程可见提供了保证，它们之间是先行发生关系。这个关系定义了一些规则让程序员在并发编程时思路更清晰。比如，先行发生关系确保了：

* 线程内的代码能够按先后顺序执行，这被称为程序次序规则。
* 对于同一个锁，一个解锁操作一定要发生在时间上后发生的另一个锁定操作之前，也叫做管程锁定规则。
* 前一个对volatile的写操作在后一个volatile的读操作之前，也叫volatile变量规则。
* 一个线程内的任何操作必需在这个线程的start()调用之后，也叫作线程启动规则。
* 一个线程的所有操作都会在线程终止之前，线程终止规则。
* 一个对象的终结操作必需在这个对象构造完成之后，也叫对象终结规则。
* 可传递性

强烈建议大家阅读《Java高并发核心编程（卷2）：多线程、锁、JMM、JUC、高并发设计模式》,来加深对Java内存模型的理解。

**9) Java中的volatile 变量是什么？**

volatile是一个特殊的修饰符，只有成员变量才能使用它。在Java并发程序缺少同步类的情况下，多线程对成员变量的操作对其它线程是透明的。volatile变量可以保证下一个读取操作会在前一个写操作之后发生。线程都会直接从内存中读取该变量并且不缓存它。这就确保了线程读取到的变量是同内存中是一致的。

**11) 什么是线程安全？Vector是一个线程安全类吗？**

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。一个线程安全的计数器类的同一个实例对象在被多个线程使用的情况下也不会出现计算失误。很显然你可以将集合类分成两组，线程安全和非线程安全的。Vector 是用同步方法来实现线程安全的, 而和它相似的ArrayList不是线程安全的。

**12) Java中什么是竞态条件？**

在大多数实际的多线程应用中，两个或两个以上的线程需要共享对同一数据的存取。如果i线程存取相同的对象，并且每一个线程都调用了一个修改该对象状态的方法，将会发生什么呢？可以想象，线程彼此踩了对方的脚。根据线程访问数据的次序，可能会产生讹误的对象。这样的情况通常称为竞争条件。

**13) Java中如何停止一个线程？**

Java提供了很丰富的API但没有为停止线程提供API。JDK 1.0本来有一些像stop(), suspend() 和 resume()的控制方法，但是由于潜在的死锁威胁。因此在后续的JDK版本中他们被弃用了，之后Java API的设计者就没有提供一个兼容且线程安全的方法来停止一个线程。当run() 或者 call() 方法执行完的时候线程会自动结束，如果要手动结束一个线程，可以用volatile 布尔变量来退出run()方法的循环或者是取消任务来中断线程。

**14) 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。

**19) 什么是FutureTask？**

在Java并发程序中FutureTask表示一个可以取消的异步运算。它有启动和取消运算、查询运算是否完成和取回运算结果等方法。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也是调用了Runnable接口所以它可以提交给Executor来执行。

**20) Java中interrupted 和 isInterruptedd方法的区别？**

interrupted() 和 isInterrupted()的主要区别是前者会将中断状态清除而后者不会。Java多线程的中断机制是用内部标识来实现的，调用Thread.interrupt()来中断一个线程就会设置中断标识为true。当中断线程调用静态方法Thread.interrupted()来检查中断状态时，中断状态会被清零。而非静态方法isInterrupted()用来查询其它线程的中断状态且不会改变中断状态标识。简单的说就是任何抛出InterruptedException异常的方法都会将中断状态清零。无论如何，一个线程的中断状态有有可能被其它线程调用中断来改变。

**23) 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。更多内容详见答案。

**24） Java中堆和栈有什么不同？**

为什么把这个问题归类在多线程和并发面试题里？因为栈是一块和线程紧密相关的内存区域。每个线程都有自己的栈内存，用于存储本地变量，方法参数和栈调用，一个线程中存储的变量对其它线程是不可见的。而堆是所有线程共享的一片公用内存区域。对象都在堆里创建，为了提升效率线程会从堆中弄一个缓存到自己的栈，如果多个线程使用该变量就可能引发问题，这时volatile 变量就可以发挥作用了，它要求线程从主存中读取变量的值。

**26） 如何写代码来解决生产者消费者问题？**

在现实中你解决的许多线程问题都属于生产者消费者模型，就是一个线程生产任务供其它线程进行消费，你必须知道怎么进行线程间通信来解决这个问题。比较低级的办法是用wait和notify来解决这个问题，比较赞的办法是用Semaphore 或者 BlockingQueue来实现生产者消费者模型。

**27） 如何避免死锁？**

Java多线程中的死锁

死锁是指两个或两个以上的进程在执行过程中，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。这是一个严重的问题，因为死锁会让你的程序挂起无法完成任务，死锁的发生必须满足以下四个条件：

* 互斥条件：一个资源每次只能被一个进程使用。
* 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。
* 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。
* 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。

避免死锁最简单的方法就是阻止循环等待条件，将系统中所有的资源设置标志位、排序，规定所有的进程申请资源必须以一定的顺序（升序或降序）做操作来避免死锁。

**28) Java中活锁和死锁有什么区别？**

这是上题的扩展，活锁和死锁类似，不同之处在于处于活锁的线程或进程的状态是不断改变的，活锁可以认为是一种特殊的饥饿。一个现实的活锁例子是两个人在狭小的走廊碰到，两个人都试着避让对方好让彼此通过，但是因为避让的方向都一样导致最后谁都不能通过走廊。简单的说就是，活锁和死锁的主要区别是前者进程的状态可以改变但是却不能继续执行。

**29） 怎么检测一个线程是否拥有锁？**

在java.lang.Thread中有一个方法叫holdsLock()，它返回true如果当且仅当当前线程拥有某个具体对象的锁。

**30) 你如何在Java中获取线程堆栈？**

对于不同的操作系统，有多种方法来获得Java进程的线程堆栈。当你获取线程堆栈时，JVM会把所有线程的状态存到日志文件或者输出到控制台。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。

**31) JVM中哪个参数是用来控制线程的栈堆栈小的**

这个问题很简单， -Xss参数用来控制线程的堆栈大小。你可以查看JVM配置列表来了解这个参数的更多信息。

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

Java在过去很长一段时间只能通过synchronized关键字来实现互斥，它有一些缺点。比如你不能扩展锁之外的方法或者块边界，尝试获取锁时不能中途取消等。Java 5 通过Lock接口提供了更复杂的控制来解决这些问题。 ReentrantLock 类实现了 Lock，它拥有与 synchronized 相同的并发性和内存语义且它还具有可扩展性。

**33） 有三个线程T1，T2，T3，怎么确保它们按顺序执行（确保main()方法所在的线程是Java程序最后结束的线程）？**

在多线程中有多种方法让线程按特定顺序执行，你可以用线程类的join()方法在一个线程中启动另一个线程，另外一个线程完成该线程继续执行。为了确保三个线程的顺序你应该先启动最后一个(T3调用T2，T2调用T1)，这样T1就会先完成而T3最后完成。

**34) 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。点击这里查看更多yield方法的相关内容。

**35） 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。

**36） Java中Semaphore是什么？**

Java中的Semaphore是一种新的同步类，它是一个计数信号。从概念上讲，从概念上讲，信号量维护了一个许可集合。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。每个 release()添加一个许可，从而可能释放一个正在阻塞的获取者。但是，不使用实际的许可对象，Semaphore只对可用许可的号码进行计数，并采取相应的行动。信号量常常用于多线程的代码中，比如数据库连接池。更多详细信息请点击这里。

**37）如果你提交任务时，线程池队列已满。会时发会生什么？**

这个问题问得很狡猾，许多程序员会认为该任务会阻塞直到线程池队列有空位。事实上如果一个任务不能被调度执行那么ThreadPoolExecutor’s submit()方法将会抛出一个RejectedExecutionException异常。

**38) Java线程池中submit() 和 execute()方法有什么区别？**

两个方法都可以向线程池提交任务，execute()方法的返回类型是void，它定义在Executor接口中, 而submit()方法可以返回持有计算结果的Future对象，它定义在ExecutorService接口中，它扩展了Executor接口，其它线程池类像ThreadPoolExecutor和ScheduledThreadPoolExecutor都有这些方法。更多详细信息请点击这里。

**39) 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。更多详细信息请点击这里。

**40） 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。

**41） 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）。

**42） 在多线程中，什么是上下文切换(context-switching)？**

上下文切换是存储和恢复CPU状态的过程，它使得线程执行能够从中断点恢复执行。上下文切换是多任务操作系统和多线程环境的基本特征。

**43) 如何在Java中创建Immutable对象？**

Immutable对象可以在没有同步的情况下共享，降低了对该对象进行并发访问时的同步化开销。要创建不可变类，要实现下面几个步骤：通过构造方法初始化所有成员、对变量不要提供setter方法、将所有的成员声明为私有的，这样就不允许直接访问这些成员、在getter方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷贝。

**44） Java中的ReadWriteLock是什么？**

一般而言，读写锁是用来提升并发程序性能的锁分离技术的成果。Java中的ReadWriteLock是Java 5 中新增的一个接口，一个ReadWriteLock维护一对关联的锁，一个用于只读操作一个用于写。在没有写线程的情况下一个读锁可能会同时被多个读线程持有。写锁是独占的，你可以使用JDK中的ReentrantReadWriteLock来实现这个规则，它最多支持65535个写锁和65535个读锁。

**45) 多线程中的忙循环是什么?**

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。

**46）volatile 变量和 atomic 变量有什么不同？**

这是个有趣的问题。首先，volatile 变量和 atomic 变量看起来很像，但功能却不一样。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。

**47) 如果同步块内的线程抛出异常会发生什么？**

这个问题坑了很多Java程序员，若你能想到锁是否释放这条线索来回答还有点希望答对。无论你的同步块是正常还是异常退出的，里面的线程都会释放锁，所以对比锁接口我们更喜欢同步块，因为它不用花费精力去释放锁，该功能可以在finally block里释放锁实现。

**48） 单例模式的双检锁是什么？**

这个问题在Java面试中经常被问到，但是面试官对回答此问题的满意度仅为50%。一半的人写不出双检锁还有一半的人说不出它的隐患和Java1.5是如何对它修正的。它其实是一个用来创建线程安全的单例的老方法，当单例实例第一次被创建时它试图用单个锁进行性能优化，但是由于太过于复杂在JDK1.4中它是失败的。

**49） 如何在Java中创建线程安全的Singleton？**

这是上面那个问题的后续，如果你不喜欢双检锁而面试官问了创建Singleton类的替代方法，你可以利用JVM的类加载和静态变量初始化特征来创建Singleton实例，或者是利用枚举类型来创建Singleton。

**50) 写出3条你遵循的多线程最佳实践**

以下三条最佳实践大多数Java程序员都应该遵循：

* 给你的线程起个有意义的名字。

这样可以方便找bug或追踪。OrderProcessor, QuoteProcessor or TradeProcessor 这种名字比 Thread-1. Thread-2 and Thread-3 好多了，给线程起一个和它要完成的任务相关的名字，所有的主要框架甚至JDK都遵循这个最佳实践。

* 避免锁定和缩小同步的范围

锁花费的代价高昂且上下文切换更耗费时间空间，试试最低限度的使用同步和锁，缩小临界区。因此相对于同步方法我更喜欢同步块，它给我拥有对锁的绝对控制权。

* 多用同步类少用wait 和 notify

首先，CountDownLatch, Semaphore, CyclicBarrier 和 Exchanger 这些同步类简化了编码操作，而用wait和notify很难实现对复杂控制流的控制。其次，这些类是由最好的企业编写和维护在后续的JDK中它们还会不断优化和完善，使用这些更高等级的同步工具你的程序可以不费吹灰之力获得优化。

* 多用并发集合少用同步集合

这是另外一个容易遵循且受益巨大的最佳实践，并发集合比同步集合的可扩展性更好，所以在并发编程时使用并发集合效果更好。如果下一次你需要用到map，你应该首先想到用ConcurrentHashMap。

**51) 如何强制启动一个线程？**

这个问题就像是如何强制进行Java垃圾回收，目前还没有觉得方法，虽然你可以使用System.gc()来进行垃圾回收，但是不保证能成功。在Java里面没有办法强制启动一个线程，它是被线程调度器控制着且Java没有公布相关的API。

**52) Java中的fork join框架是什么？**

fork join框架是JDK7中出现的一款高效的工具，Java开发人员可以通过它充分利用现代服务器上的多处理器。它是专门为了那些可以递归划分成许多子模块设计的，目的是将所有可用的处理能力用来提升程序的性能。fork join框架一个巨大的优势是它使用了工作窃取算法，可以完成更多任务的工作线程可以从其它线程中窃取任务来执行。

**53） Java多线程中调用wait() 和 sleep()方法有什么不同？**

Java程序中wait 和 sleep都会造成某种形式的暂停，它们可以满足不同的需要。wait()方法用于线程间通信，如果等待条件为真且其它线程被唤醒时它会释放锁，而sleep()方法仅仅释放CPU资源或者让当前线程停止执行一段时间，但不会释放锁。需要注意的是，sleep（）并不会让线程终止，一旦从休眠中唤醒线程，线程的状态将会被改变为Runnable，并且根据线程调度，它将得到执行。

**54） 什么是Thread Group？为什么不建议使用它？**

ThreadGroup是一个类，它的目的是提供关于线程组的信息。

ThreadGroup API比较薄弱，它并没有比Thread提供了更多的功能。它有两个主要的功能：一是获取线程组中处于活跃状态线程的列表；二是设置为线程设置未捕获异常处理器(ncaught exception handler)。但在Java 1.5中Thread类也添加了setUncaughtExceptionHandler(UncaughtExceptionHandler eh) 方法，所以ThreadGroup是已经过时的，不建议继续使用。

**55) 什么是Java线程转储(Thread Dump)，如何得到它？**

线程转储是一个JVM活动线程的列表，它对于分析系统瓶颈和死锁非常有用。有很多方法可以获取线程转储——使用Profiler，Kill -3命令，jstack工具等等。我们更喜欢jstack工具，因为它容易使用并且是JDK自带的。由于它是一个基于终端的工具，所以我们可以编写一些脚本去定时的产生线程转储以待分析。

**56) 什么是Java Timer类？如何创建一个有特定时间间隔的任务？**

java.util.Timer是一个工具类，可以用于安排一个线程在未来的某个特定时间执行。Timer类可以用安排一次性任务或者周期任务。

java.util.TimerTask是一个实现了Runnable接口的抽象类，我们需要去继承这个类来创建我们自己的定时任务并使用Timer去安排它的执行。

**57) 什么是原子操作？在Java Concurrency API中有哪些原子类(atomic classes)？**

原子操作是指一个不受其他操作影响的操作任务单元。原子操作是在多线程环境下避免数据不一致必须的手段。

int++并不是一个原子操作，所以当一个线程读取它的值并加1时，另外一个线程有可能会读到之前的值，这就会引发错误。

在 java.util.concurrent.atomic 包中添加原子变量类之后，这种情况才发生了改变。所有原子变量类都公开比较并设置原语（与比较并交换类似），这些原语都是使用平台上可用的最快本机结构（比较并交换、加载链接/条件存储，最坏的情况下是旋转锁）来实现的。 java.util.concurrent.atomic 包中提供了原子变量的 9 种风格（ AtomicInteger； AtomicLong； AtomicReference； AtomicBoolean；原子整型；长型；引用；及原子标记引用和戳记引用类的数组形式，其原子地更新一对值）。

**58） Java Concurrency API中的Lock接口(Lock interface)是什么？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：

* 可以使锁更公平
* 可以使线程在等待锁的时候响应中断
* 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
* 可以在不同的范围，以不同的顺序获取和释放锁

**62）什么是Callable和Future?**

Java 5在concurrency包中引入了java.util.concurrent.Callable 接口，它和Runnable接口很相似，但它可以返回一个对象或者抛出一个异常。

Callable接口使用泛型去定义它的返回类型。Executors类提供了一些有用的方法去在线程池中执行Callable内的任务。由于Callable任务是并行的，我们必须等待它返回的结果。java.util.concurrent.Future对象为我们解决了这个问题。在线程池提交Callable任务后返回了一个Future对象，使用它我们可以知道Callable任务的状态和得到Callable返回的执行结果。Future提供了get()方法让我们可以等待Callable结束并获取它的执行结果。

**63） 什么是FutureTask?**

FutureTask包装器是一种非常便利的机制，可将Callable转换成Future和Runnable，它同时实现两者的接口。

FutureTask类是Future 的一个实现，并实现了Runnable，所以可通过Excutor(线程池) 来执行。也可传递给Thread对象执行。如果在主线程中需要执行比较耗时的操作时，但又不想阻塞主线程时，可以把这些作业交给Future对象在后台完成，当主线程将来需要时，就可以通过Future对象获得后台作业的计算结果或者执行状态。

**64） 什么是并发容器的实现？**

Java集合类都是快速失败的，这就意味着当集合被改变且一个线程在使用迭代器遍历集合的时候，迭代器的next()方法将抛出ConcurrentModificationException异常。

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

**65、如何创建守护线程？**

使用Thread类的setDaemon(true)方法可以将线程设置为守护线程，需要注意的是，需要在调用start()方法前调用这个方法，否则会抛出IllegalThreadStateException异常。

**65）用户线程和守护线程有什么区别？**

当我们在Java程序中创建一个线程，它就被称为用户线程。一个守护线程是在后台执行并且不会阻止JVM终止的线程。当没有用户线程在运行的时候，JVM关闭程序并且退出。一个守护线程创建的子线程依然是守护线程。

**66）有哪些不同的线程生命周期？**

当我们在Java程序中新建一个线程时，它的状态是New。当我们调用线程的start()方法时，状态被改变为Runnable。线程调度器会为Runnable线程池中的线程分配CPU时间并且讲它们的状态改变为Running。其他的线程状态还有Waiting，Blocked 和Dead。

**67）线程之间是如何通信的？**

当线程间是可以共享资源时，线程间通信是协调它们的重要的手段。Object类中wait()\notify()\notifyAll()方法可以用于线程间通信关于资源的锁的状态。

**68）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。

**69、如何确保线程安全？**

在Java中可以有很多方法来保证线程安全——

* 同步，
* 使用原子类(atomic concurrent classes)，
* 使用显示锁，
* 使用volatile关键字，
* 使用不变类

**72、线程调度策略？**

(1) 抢占式调度策略

Java运行时系统的线程调度算法是抢占式的 (preemptive)。Java运行时系统支持一种简单的固定优先级的调度算法。如果一个优先级比其他任何处于可运行状态的线程都高的线程进入就绪状态，那么运行时系统就会选择该线程运行。新的优先级较高的线程抢占(preempt)了其他线程。但是Java运行时系统并不抢占同优先级的线程。换句话说，Java运行时系统不是分时的(time-slice)。然而，基于Java Thread类的实现系统可能是支持分时的，因此编写代码时不要依赖分时。当系统中的处于就绪状态的线程都具有相同优先级时，线程调度程序采用一种简单的、非抢占式的轮转的调度顺序。

(2) 时间片轮转调度策略

有些系统的线程调度采用时间片轮转(round-robin)调度策略。这种调度策略是从所有处于就绪状态的线程中选择优先级最高的线程分配一定的CPU时间运行。该时间过后再选择其他线程运行。只有当线程运行结束、放弃(yield)CPU或由于某种原因进入阻塞状态，低优先级的线程才有机会执行。如果有两个优先级相同的线程都在等待CPU，则调度程序以轮转的方式选择运行的线程。

**73、 在线程中你怎么处理不可捕捉异常？**

Thread.UncaughtExceptionHandler是java SE5中的新接口，它允许我们在每一个Thread对象上添加一个异常处理器。

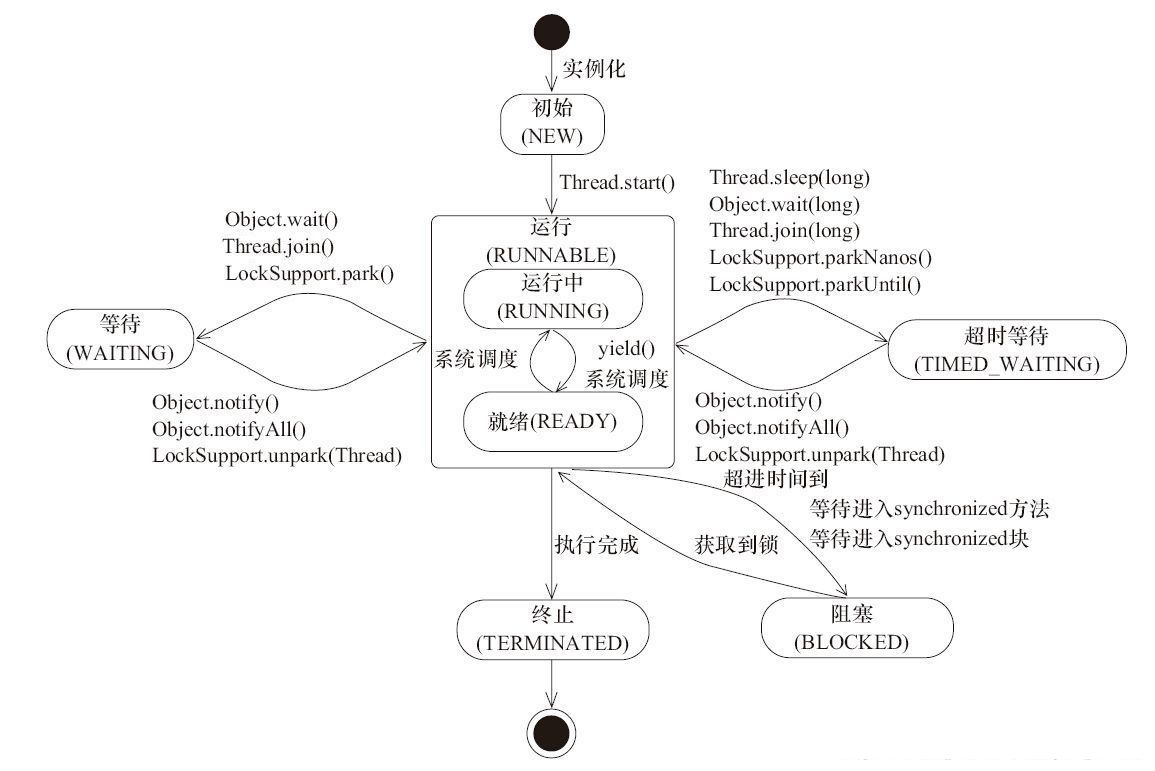
**二：Java线程的6种状态及切换**

**1、Java中线程的状态有哪些？**

Java中线程的状态分为6种。

1. **初始(NEW)**：新创建了一个线程对象，但还没有调用start()方法。
2. **运行(RUNNABLE)**：Java线程中将就绪（ready）和运行中（running）两种状态笼统的称为“运行”。  
   线程对象创建后，其他线程(比如main线程）调用了该对象的start()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取CPU的使用权，此时处于就绪状态（ready）。就绪状态的线程在获得CPU时间片后变为运行中状态（running）。
3. **阻塞(BLOCKED)**：表示线程阻塞于锁。
4. **等待(WAITING)**：进入该状态的线程需要等待其他线程做出一些特定动作（通知或中断）。
5. **超时等待(TIMED\_WAITING)**：该状态不同于WAITING，它可以在指定的时间后自行返回。
6. **终止(TERMINATED)**：表示该线程已经执行完毕。

这6种状态定义在Thread类的State枚举中，可查看源码进行一一对应。

**线程的状态图 [](https://img-blog.csdnimg.cn/20181120173640764.jpeg)**

**状态详细说明**

**1. 初始状态(NEW)**

实现Runnable接口和继承Thread可以得到一个线程类，new一个实例出来，线程就进入了初始状态。

**2.执行Thread#start之后，线程进行RUNNABLE可运行状态**

**2.1. 就绪状态(RUNNABLE之READY)**

1. 就绪状态只是说你资格运行，调度程序没有挑选到你，你就永远是就绪状态。
2. 调用线程的start()方法，此线程进入就绪状态。
3. 当前线程sleep()方法结束，其他线程join()结束，等待用户输入完毕，某个线程拿到对象锁，这些线程也将进入就绪状态。
4. 当前线程时间片用完了，调用当前线程的yield()方法，当前线程进入就绪状态。
5. 锁池里的线程拿到对象锁后，进入就绪状态。

**2.2. 运行中状态(RUNNABLE之RUNNING)**

线程调度程序从可运行池中选择一个线程作为当前线程时线程所处的状态。这也是线程进入运行状态的唯一的一种方式。

**3. 阻塞状态(BLOCKED)**

阻塞状态是线程阻塞在进入synchronized关键字修饰的方法或代码块(获取锁)时的状态。

**4. 等待(WAITING)**

处于这种状态的线程不会被分配CPU执行时间，它们要等待被显式地唤醒，否则会处于无限期等待的状态。

**5. 超时等待(TIMED\_WAITING)**

处于这种状态的线程不会被分配CPU执行时间，不过无须无限期等待被其他线程显示地唤醒，在达到一定时间后它们会自动唤醒。

**6. 终止状态(TERMINATED)**

1. 当线程的run()方法完成时，或者主线程的main()方法完成时，我们就认为它终止了。这个线程对象也许是活的，但是它已经不是一个单独执行的线程。线程一旦终止了，就不能复生。
2. 在一个终止的线程上调用start()方法，会抛出java.lang.IllegalThreadStateException异常。

**三：synchronized同步面试题**

**1. synchronized的简介**

**synchronized**要理解为**加锁**，而不是锁，这个思维有助于你更好的理解线程同步。

**1.1 synchronized的使用及各自的锁对象**

这里简要介绍一下，为以后的内容做一下铺垫：

1. 普通方法 ：锁对象是this，所谓的**方法锁**（本质上属于对象锁)

public synchronized void say(){

System.out.println("Hello,everyone...");

}

1. 同步代码块(**方法中**)：锁对象是synchronized(obj)的对象，所谓的**对象锁**

public void say(boolean isYou){

synchronized (obj){

System.out.println("Hello");

}

}

1. 同步静态方法：锁对象是当前类的Class对象,即(XXX.class)，所谓的**类锁**

public static synchronized void work(){

System.out.println("Work hard...");

}

希望大家再遇到**对象锁**,**类锁**而不知所措…有时候遇到面试官把问题描述的不够清楚时，要勇于及时和面试官沟通。虽然找工作时总会遇到奇葩面试官，但是如果你遇到的几率太高时，请自觉地审视一下自己…

**1.2 synchronized的关于代码块的疑问**

看到上面synchronized的用法，你会有这样的疑问吗？synchronized能修饰**类级别(静态)代码块**吗？(PS: 这个话题是我临时想起的,改天在面试中问一下看看效果…)

结论：synchronized不能用在**类级别的(静态)代码块**

如果在面试中不给你编译器，大多数人估计都是要mountain泰吧。这里直接给出我的理解：

这个要从加载顺序上考虑。  
类级别的代码块在加载顺序上是要优先于任何方法的，其执行顺序只跟代码位置先后有关。没人跟你抢，自然不需要同步。

**2. 涉及synchronized的面试题**

这里通过一个常见的面试题-**单例模式**来展开，这篇文章主要内容是考察synchronized关键字的，就直奔主题进入**DCL(Double Check Lock)双重校验锁**的单例。

**2.1 围绕着DCL展开的话题**

2.1.1 实现DCL

如果DCL不懂，就尴尬无止境啦…。感兴趣的可以去看一下[《Java高并发核心编程（卷2）》](https://www.cnblogs.com/crazymakercircle/p/9904544.html)

public class SingleInstance {

private volatile static SingleInstance instance = null;

private SingleInstance(){ }

public static SingleInstance getInstance(){

if (instance == null){

synchronized (SingleInstance.class){

if (instance == null){

instance = new SingleInstance();

}

}

}

return instance;

}

}

2.1.2 谈一下synchronized的作用

synchronized 关键字主要用来解决的是多线程同步问题，其可以保证在被其修饰的代码任意时刻只有一个线程执行。视情况而定，（主动)说出它的用法及底层实现原理(使用的是moniterenter 和 moniterexit指令…)，PS：synchronized的底层实现原理会单独展开…

2.1.3 这里(DCL)的volatile的作用

volatile只能保证变量的可见性，并不能保证对volatile修饰的变量的操作的原子性。

volatile的主要作用：

1. 保持内存可见性；使所有线程都能看到共享内存的最新状态。
2. 防止指令重排的问题；

通过设置**内存屏障**实现的。感兴趣的可以去看一下[《Java高并发核心编程（卷2）》](https://www.cnblogs.com/crazymakercircle/p/9904544.html)

个人拙见，能答出来上面的内容即可，更深入的绝大部分都是在SHOW或者就是压薪资…

**2.2 基础面试点**

为了节约各位看官的时间，先把结论给出来：

1. 若是对象锁，则每个对象都持有一把自己的独一无二的锁，且对象之间的锁互不影响 。若是类锁，所有该类的对象共用这把锁。
2. 一个线程获取一把锁，没有得到锁的线程只能排队等待；
3. synchronized 是可重入锁，避免很多情况下的死锁发生。
4. synchronized 方法若发生异常，则JVM会自动释放锁。
5. 锁对象不能为空，否则抛出NPE(NullPointerException)
6. 同步本身是不具备继承性的：即父类的synchronized 方法，子类重写该方法,分情况讨论：没有synchonized修饰，则该子类方法不是线程同步的。(PS ：涉及同步继承性的问题要分情况)
7. synchronized本身修饰的范围越小越好。毕竟是同步阻塞。跑不快还占着超车道…

2.2.1 同时访问synchronized的静态和非静态方法，能保证线程安全吗？

结论：不能，两者的锁对象不一样。前者是类锁(XXX.class),后者是this

2.2.2 同时访问synchronized方法和非同步方法，能保证线程安全吗？

结论：不能，因为synchronized只会对被修饰的方法起作用。

2.2.3 两个线程同时访问两个对象的非静态同步方法能保证线程安全吗？

结论：不能，每个对象都拥有一把锁。两个对象相当于有两把锁，导致锁对象不一致。(PS：如果是类锁，则所有对象共用一把锁)

2.2.4 若synchronized方法抛出异常，会导致死锁吗？

JVM会自动释放锁，不会导致死锁问题

2.2.5 若synchronized的锁对象能为空吗？会出现什么情况？

锁对象不能为空，否则抛出NPE(NullPointerException)

2.2.6 若synchronized的锁对象能为空吗？会出现什么情况？

锁对象不能为空，否则抛出NPE(NullPointerException)

**2.3 关于继承性的面试点**

2.3.1 synchronized涉及的继承性问题

重写父类的synchronized的方法，主要分为两种情况：

1. 子类的方法没有被synchronized修饰：

synchronized的不具备继承性。所以子类方法是线程不安全的。

1. 子类的方法被synchronized修饰(这里面试点主要考察锁对象的归属问题)：

两个锁对象其实是一把锁，而且是**子类对象作为锁**。这也证明了: synchronized的锁是可重入锁。否则将出现死锁问题。

**2.4 实战经验的面试点**

2.4.1 在开发过程中，你经常使用synchronized方法多还是synchronized代码块？and why?

关于synchronized 的内容部分，我在面试过程中经常问且只问这一道题。本人认为这个能很好的考察面试者的综合素质。(PS：毕竟是要拧螺丝的…)

synchronized同步的范围是越小越好。因为若该方法耗时很久，那其它线程必须等到该持锁线程执行完才能运行。(黄花菜都凉了都…)  
而synchronized代码块部分只有这一部分是同步的，其它的照样可以异步执行，提高运行效率。

**3、对象锁的同步队列**

* 当前线程想调用对象A的同步方法时，发现对象A的锁被别的线程占有，此时当前线程进入对象锁的同步队列。简言之，同步队列里面放的都是想争夺对象锁的线程。
* 当一个线程1被另外一个线程2唤醒时，1线程进入同步队列，去争夺对象锁。
* 同步队列是在同步的环境下才有的概念，一个对象对应一个同步队列。
* 线程等待时间到了或被notify/notifyAll唤醒后，会进入同步队列竞争锁，如果获得锁，进入RUNNABLE状态，否则进入BLOCKED状态等待获取锁。

**4、 如何在两个线程间共享数据？**

你可以通过共享对象来实现这个目的，或者是使用像阻塞队列这样并发的数据结构。这篇教程《Java线程间通信》(涉及到在两个线程间共享对象)用wait和notify方法实现了生产者消费者模型。

**5、 Java中notify 和 notifyAll有什么区别？**

这又是一个刁钻的问题，因为多线程可以等待单监控锁，Java API 的设计人员提供了一些方法当等待条件改变的时候通知它们，但是这些方法没有完全实现。notify()方法不能唤醒某个具体的线程，所以只有一个线程在等待的时候它才有用武之地。而notifyAll()唤醒所有线程并允许他们争夺锁确保了至少有一个线程能继续运行。

**6) 为什么wait, notify 和 notifyAll这些方法不在thread类里面？**

一个很明显的原因是JAVA提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线程获得。如果线程需要等待某些锁那么调用对象中的wait()方法就有意义了。如果wait()方法定义在Thread类中，线程正在等待的是哪个锁就不明显了。简单的说，由于wait，notify和notifyAll都是锁级别的操作，所以把他们定义在Object类中因为锁属于对象。

**7) 为什么wait和notify方法要在同步块中调用？**

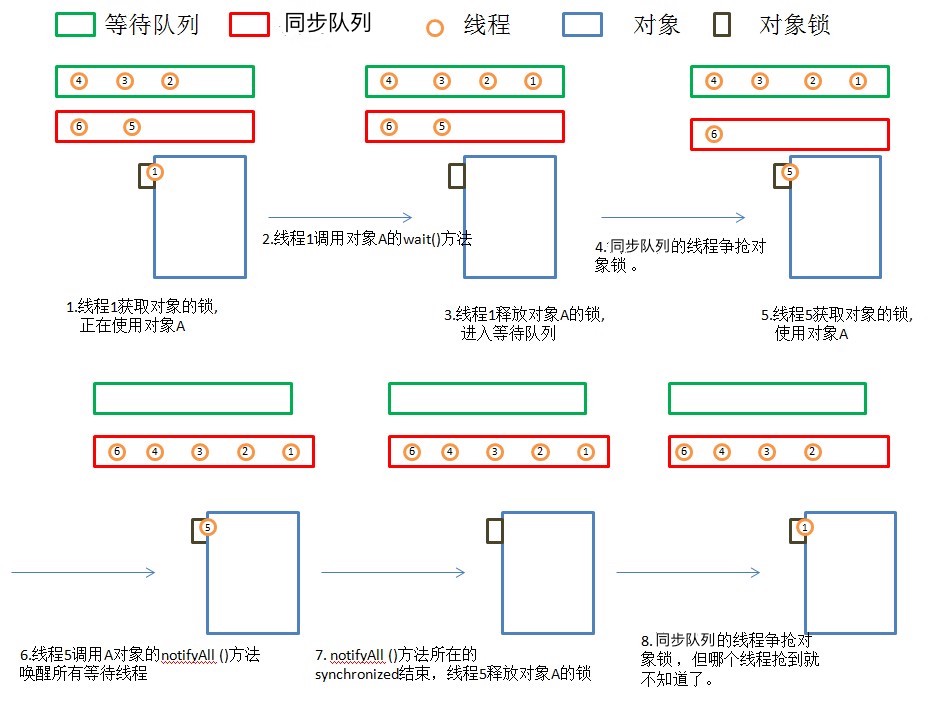
当一个线程需要调用对象的wait()方法的时候，这个线程必须拥有该对象的锁，接着它就会释放这个对象锁并进入等待状态直到其他线程调用这个对象上的notify()方法。同样的，当一个线程需要调用对象的notify()方法时，它会释放这个对象的锁，以便其他在等待的线程就可以得到这个对象锁。由于所有的这些方法都需要线程持有对象的锁，这样就只能通过同步来实现，所以他们只能在同步方法或者同步块中被调用。如果你不这么做，代码会抛出IllegalMonitorStateException异常。

**8) 为什么应该在循环中检查等待条件?**

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。因此，当一个等待线程醒来时，不能认为它原来的等待状态仍然是有效的，在notify()方法调用之后和等待线程醒来之前这段时间它可能会改变。这就是在循环中使用wait()方法效果更好的原因，你可以在Eclipse中创建模板调用wait和notify试一试。如果你想了解更多关于这个问题的内容，推荐你阅读《Effective Java》这本书中的线程和同步章节。

**9、对象锁的等待队列**

* 调用obj的wait(), notify()方法前，必须获得obj锁，也就是必须写在synchronized(obj) 代码段内。
* 与等待队列相关的步骤和图

[](https://img-blog.csdn.net/20180701221233161)

1. 线程1获取对象A的锁，正在使用对象A。
2. 线程1调用对象A的wait()方法。
3. 线程1释放对象A的锁，并马上进入等待队列。
4. 锁池里面的对象争抢对象A的锁。
5. 线程5获得对象A的锁，进入synchronized块，使用对象A。
6. 线程5调用对象A的notifyAll()方法，唤醒所有线程，所有线程进入同步队列。若线程5调用对象A的notify()方法，则唤醒一个线程，不知道会唤醒谁，被唤醒的那个线程进入同步队列。
7. notifyAll()方法所在synchronized结束，线程5释放对象A的锁。
8. 同步队列的线程争抢对象锁，但线程1什么时候能抢到就不知道了。

**四、深入分析Synchronized原理(阿里面试题)**

记得开始学习Java的时候，一遇到多线程情况就使用synchronized，相对于当时的我们来说synchronized是这么的神奇而又强大，那个时候我们赋予它一个名字“同步”，也成为了我们解决多线程情况的百试不爽的良药。但是，随着学习的进行我们知道在JDK1.5之前synchronized是一个重量级锁，相对于j.u.c.Lock，它会显得那么笨重，以至于我们认为它不是那么的高效而慢慢摒弃它。

不过，随着Javs SE 1.6对synchronized进行的各种优化后，synchronized并不会显得那么重了。下面来一起探索synchronized的基本使用、实现机制、Java是如何对它进行了优化、锁优化机制、锁的存储结构等升级过程。

**1 基本使用**

Synchronized是Java中解决并发问题的一种最常用的方法，也是最简单的一种方法。Synchronized的作用主要有三个：

1. 原子性：确保线程互斥的访问同步代码；
2. 可见性：保证共享变量的修改能够及时可见，其实是通过Java内存模型中的 “**对一个变量unlock操作之前，必须要同步到主内存中；如果对一个变量进行lock操作，则将会清空工作内存中此变量的值，在执行引擎使用此变量前，需要重新从主内存中load操作或assign操作初始化变量值**” 来保证的；
3. 有序性：有效解决重排序问题，即 “一个unlock操作先行发生(happen-before)于后面对同一个锁的lock操作”；

从语法上讲，Synchronized可以把任何一个非null对象作为"锁"，在HotSpot JVM实现中，**锁有个专门的名字：对象监视器（Object Monitor）**。

Synchronized总共有三种用法：

1. 当synchronized作用在实例方法时，监视器锁（monitor）便是对象实例（this）；
2. 当synchronized作用在静态方法时，监视器锁（monitor）便是对象的Class实例，因为Class数据存在于永久代，因此静态方法锁相当于该类的一个全局锁；
3. 当synchronized作用在某一个对象实例时，监视器锁（monitor）便是括号括起来的对象实例；

注意，synchronized 内置锁 是一种 对象锁（锁的是对象而非引用变量），**作用粒度是对象 ，可以用来实现对 临界资源的同步互斥访问 ，是 可重入 的。其可重入最大的作用是避免死锁**，如：

**子类同步方法调用了父类同步方法，如没有可重入的特性，则会发生死锁；**

**2 同步原理**

数据同步需要依赖锁，那锁的同步又依赖谁？**synchronized给出的答案是在软件层面依赖JVM，而j.u.c.Lock给出的答案是在硬件层面依赖特殊的CPU指令。**

当一个线程访问同步代码块时，首先是需要得到锁才能执行同步代码，当退出或者抛出异常时必须要释放锁，那么它是如何来实现这个机制的呢？我们先看一段简单的代码：

package com.paddx.test.concurrent;

public class SynchronizedDemo {

public void method() {

synchronized (this) {

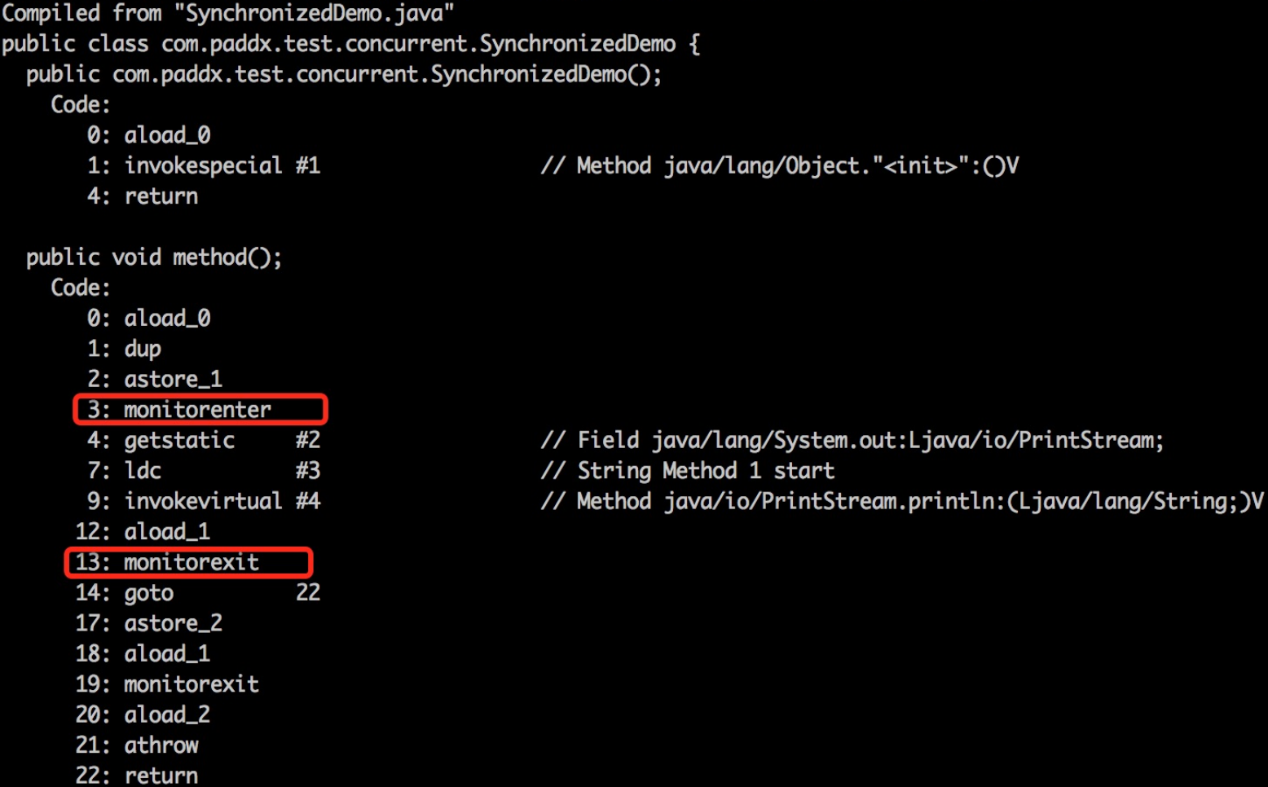
System.out.println("Method 1 start");

}

}

}

查看反编译后结果：

[](https://img-blog.csdnimg.cn/img_convert/98d26e44b0a41e2de06fd86633f356c3.png)

反编译结果

1. **monitorenter**：每个对象都是一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：
   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；
2. monitorexit：执行monitorexit的线程必须是objectref所对应的monitor的所有者。指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

monitorexit指令出现了两次，第1次为同步正常退出释放锁；第2次为发生异步退出释放锁；

通过上面两段描述，我们应该能很清楚的看出Synchronized的实现原理，**Synchronized的语义底层是通过一个monitor的对象来完成，其实wait/notify等方法也依赖于monitor对象，这就是为什么只有在同步的块或者方法中才能调用wait/notify等方法，否则会抛出java.lang.IllegalMonitorStateException的异常的原因。**

再来看一下同步方法：

package com.paddx.test.concurrent;

public class SynchronizedMethod {

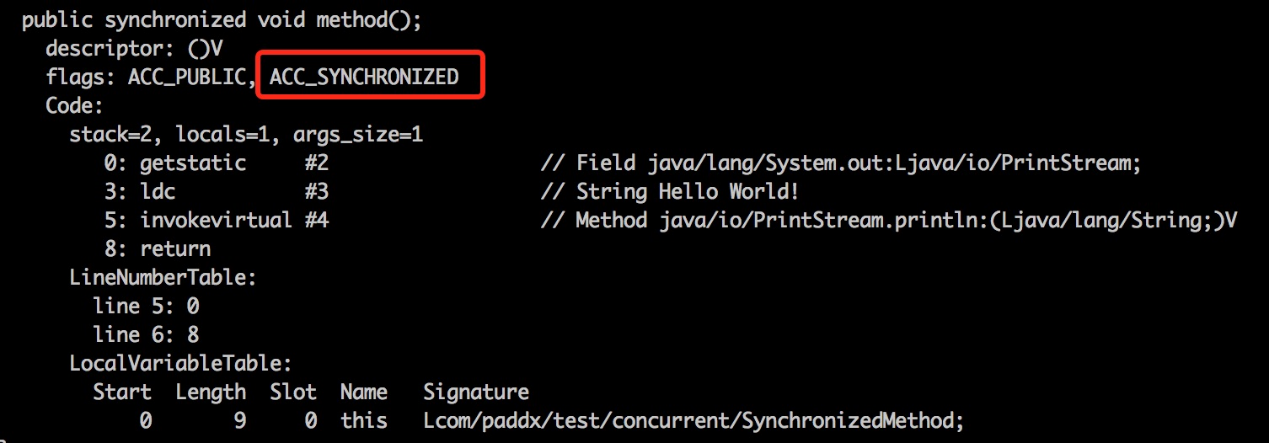
public synchronized void method() {

System.out.println("Hello World!");

}

}

查看反编译后结果：

[](https://img-blog.csdnimg.cn/img_convert/bfa1e6325e2de6806cbabdcbfe6c4fd8.png)

反编译结果

从编译的结果来看，方法的同步并没有通过指令 **monitorenter** 和 **monitorexit** 来完成（理论上其实也可以通过这两条指令来实现），不过相对于普通方法，其常量池中多了 **ACC\_SYNCHRONIZED** 标示符。JVM就是根据该标示符来实现方法的同步的：

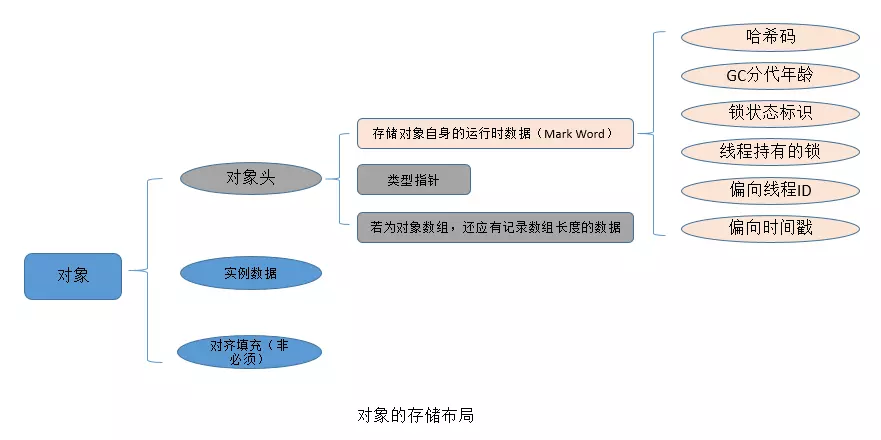
当方法调用时，调用指令将会检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。**在方法执行期间，其他任何线程都无法再获得同一个monitor对象。**

两种同步方式本质上没有区别，只是方法的同步是一种隐式的方式来实现，无需通过字节码来完成。两个指令的执行是JVM通过调用操作系统的互斥原语mutex来实现，被阻塞的线程会被挂起、等待重新调度，会导致“用户态和内核态”两个态之间来回切换，对性能有较大影响。

**3 同步概念**

**3.1 Java对象头**

在JVM中**，对象在内存中的布局分为三块区域：对象头、实例数据和对齐填充。**如下图所示：

[](https://img-blog.csdnimg.cn/img_convert/7ad6ab858b2d759bc1be9c5852ed4704.png)

1. 实例数据：存放类的属性数据信息，包括父类的属性信息；
2. 对齐填充：由于虚拟机要求 对象起始地址必须是8字节的整数倍。填充数据不是必须存在的，仅仅是为了字节对齐；
3. **对象头：Java对象头一般占有2个机器码（在32位虚拟机中，1个机器码等于4字节，也就是32bit，在64位虚拟机中，1个机器码是8个字节，也就是64bit），但是 如果对象是数组类型，则需要3个机器码，因为JVM虚拟机可以通过Java对象的元数据信息确定Java对象的大小，但是无法从数组的元数据来确认数组的大小，所以用一块来记录数组长度。**

Synchronized用的锁就是存在Java对象头里的，那么什么是Java对象头呢？Hotspot虚拟机的对象头主要包括两部分数据：**Mark Word（标记字段）、**Class Pointer（类型指针）。其中 Class Pointer是对象指向它的类元数据的指针，虚拟机通过这个指针来确定这个对象是哪个类的实例，Mark Word用于存储对象自身的运行时数据，它是实现轻量级锁和偏向锁的关键。 Java对象头具体结构描述如下：

[](https://img-blog.csdnimg.cn/img_convert/f8f5088839a9f1e36a3dd84e80c572e3.png)

Java对象头结构组成

Mark Word用于存储对象自身的运行时数据，如：哈希码（HashCode）、GC分代年龄、**锁状态标志**、线程持有的锁、偏向线程 ID、偏向时间戳等。比如锁膨胀就是借助Mark Word的偏向的线程ID 参考：[JAVA锁的膨胀过程和优化(阿里)](https://www.cnblogs.com/aspirant/p/11705068.html) 阿里也经常问的问题

下图是Java对象头 无锁状态下Mark Word部分的存储结构（32位虚拟机）：

[](https://img-blog.csdnimg.cn/img_convert/62321275d27c4b099fdb87e1b73419e4.png)

Mark Word存储结构

对象头信息是与对象自身定义的数据无关的额外存储成本，但是考虑到虚拟机的空间效率，Mark Word被设计成一个非固定的数据结构以便在极小的空间内存存储尽量多的数据，它会根据对象的状态复用自己的存储空间，也就是说，Mark Word会随着程序的运行发生变化，可能变化为存储以下4种数据：

[](https://img-blog.csdnimg.cn/img_convert/e05394f5d671a8ce2fc87cfa3a51d632.png)

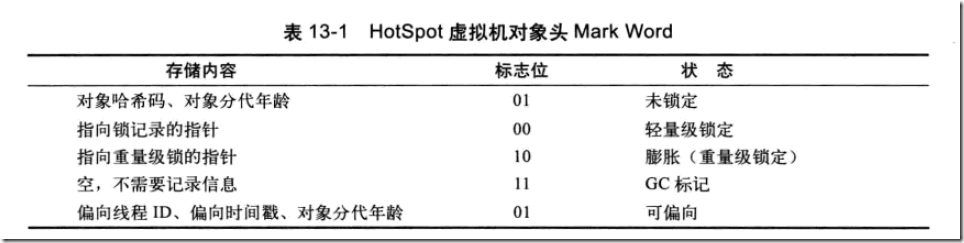
Mark Word可能存储4种数据

在64位虚拟机下，Mark Word是64bit大小的，其存储结构如下：

[](https://img-blog.csdnimg.cn/img_convert/957082b8848872abf1e3e242de3c71db.png)

64位Mark Word存储结构

对象头的最后两位存储了锁的标志位，01是初始状态，未加锁，其对象头里存储的是对象本身的哈希码，随着锁级别的不同，对象头里会存储不同的内容。偏向锁存储的是当前占用此对象的线程ID；而轻量级则存储指向线程栈中锁记录的指针。从这里我们可以看到，“锁”这个东西，可能是个锁记录+对象头里的引用指针（判断线程是否拥有锁时将线程的锁记录地址和对象头里的指针地址比较)，也可能是对象头里的线程ID（判断线程是否拥有锁时将线程的ID和对象头里存储的线程ID比较）。

[](https://img-blog.csdnimg.cn/img_convert/f2824d5fdcf5487772287bb25f078205.png)

HotSpot虚拟机对象头Mark Word

**3.2 对象头中Mark Word与线程中Lock Record**

在线程进入同步代码块的时候，如果此同步对象没有被锁定，即它的锁标志位是01，则虚拟机首先在当前线程的栈中创建我们称之为“锁记录（Lock Record）”的空间，用于存储锁对象的Mark Word的拷贝，官方把这个拷贝称为Displaced Mark Word。整个Mark Word及其拷贝至关重要。

**Lock Record是线程私有的数据结构**，每一个线程都有一个可用Lock Record列表，同时还有一个全局的可用列表。每一个被锁住的对象Mark Word都会和一个Lock Record关联（对象头的MarkWord中的Lock Word指向Lock Record的起始地址），同时Lock Record中有一个Owner字段存放拥有该锁的线程的唯一标识（或者**object mark word**），表示该锁被这个线程占用。如下图所示为Lock Record的内部结构：

| **Lock Record** | **描述** |
| --- | --- |
| Owner | 初始时为NULL表示当前没有任何线程拥有该monitor record，当线程成功拥有该锁后保存线程唯一标识，当锁被释放时又设置为NULL； |
| EntryQ | 关联一个系统互斥锁（semaphore），阻塞所有试图锁住monitor record失败的线程； |
| RcThis | 表示blocked或waiting在该monitor record上的所有线程的个数； |
| Nest | 用来实现 重入锁的计数； |
| HashCode | 保存从对象头拷贝过来的HashCode值（可能还包含GC age）。 |
| Candidate | 用来避免不必要的阻塞或等待线程唤醒，因为每一次只有一个线程能够成功拥有锁，如果每次前一个释放锁的线程唤醒所有正在阻塞或等待的线程，会引起不必要的上下文切换（从阻塞到就绪然后因为竞争锁失败又被阻塞）从而导致性能严重下降。Candidate只有两种可能的值0表示没有需要唤醒的线程1表示要唤醒一个继任线程来竞争锁。 |

**3.3 监视器（Monitor）**

任何一个对象都有一个Monitor与之关联，当且一个Monitor被持有后，它将处于锁定状态。Synchronized在JVM里的实现都是 基于进入和退出Monitor对象来实现方法同步和代码块同步，虽然具体实现细节不一样，但是都可以通过成对的MonitorEnter和MonitorExit指令来实现。

1. **MonitorEnter指令：插入在同步代码块的开始位置，当代码执行到该指令时，将会尝试获取该对象Monitor的所有权，即尝试获得该对象的锁；**
2. **MonitorExit指令：插入在方法结束处和异常处，JVM保证每个MonitorEnter必须有对应的MonitorExit；**

那什么是Monitor？可以把它理解为 一个同步工具，也可以描述为 一种同步机制，它通常被 描述为一个对象。

与一切皆对象一样，所有的Java对象是天生的Monitor，每一个Java对象都有成为Monitor的潜质，因为在Java的设计中 ，**每一个Java对象自打娘胎里出来就带了一把看不见的锁，它叫做内部锁或者Monitor锁**。

也就是通常说Synchronized的对象锁，MarkWord锁标识位为10，其中指针指向的是Monitor对象的起始地址。在Java虚拟机（HotSpot）中，Monitor是由ObjectMonitor实现的，其主要数据结构如下（位于HotSpot虚拟机源码ObjectMonitor.hpp文件，C++实现的）：

ObjectMonitor() {

\_header = NULL;

\_count = 0; // 记录个数

\_waiters = 0,

\_recursions = 0;

\_object = NULL;

\_owner = NULL;

\_WaitSet = NULL; // 处于wait状态的线程，会被加入到\_WaitSet

\_WaitSetLock = 0 ;

\_Responsible = NULL ;

\_succ = NULL ;

\_cxq = NULL ;

FreeNext = NULL ;

\_EntryList = NULL ; // 处于等待锁block状态的线程，会被加入到该列表

\_SpinFreq = 0 ;

\_SpinClock = 0 ;

OwnerIsThread = 0 ;

}

ObjectMonitor中有两个队列，\_WaitSet 和 \_EntryList，用来保存ObjectWaiter对象列表（ 每个等待锁的线程都会被封装成ObjectWaiter对象 ），\_owner指向持有ObjectMonitor对象的线程，当多个线程同时访问一段同步代码时：

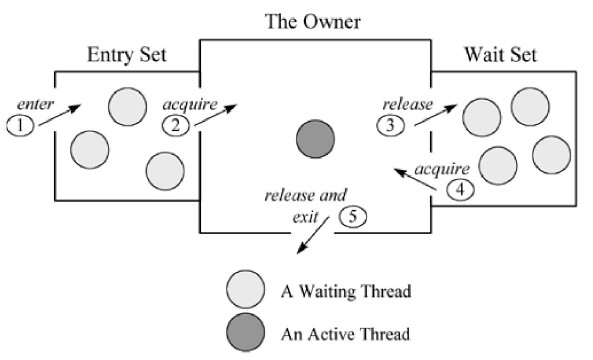
1. 首先会进入 \_EntryList 集合，当线程获取到对象的monitor后，进入 \_Owner区域并把monitor中的owner变量设置为当前线程，同时monitor中的计数器count加1；
2. 若线程调用 wait() 方法，将释放当前持有的monitor，owner变量恢复为null，count自减1，同时该线程进入 WaitSet集合中等待被唤醒；
3. 若当前线程执行完毕，也将释放monitor（锁）并复位count的值，以便其他线程进入获取monitor(锁)；

同时**，Monitor对象存在于每个Java对象的对象头Mark Word中（存储的指针的指向），Synchronized锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因，同时notify/notifyAll/wait等方法会使用到Monitor锁对象，所以必须在同步代码块中使用。**

监视器Monitor有两种同步方式：互斥与协作。多线程环境下线程之间如果需要共享数据，需要解决互斥访问数据的问题，监视器可以确保监视器上的数据在同一时刻只会有一个线程在访问。

什么时候需要协作？ 比如：

一个线程向缓冲区写数据，另一个线程从缓冲区读数据，如果读线程发现缓冲区为空就会等待，当写线程向缓冲区写入数据，就会唤醒读线程，这里读线程和写线程就是一个合作关系。JVM通过Object类的wait方法来使自己等待，在调用wait方法后，该线程会释放它持有的监视器，直到其他线程通知它才有执行的机会。一个线程调用notify方法通知在等待的线程，这个等待的线程并不会马上执行，而是要通知线程释放监视器后，它重新获取监视器才有执行的机会。如果刚好唤醒的这个线程需要的监视器被其他线程抢占，那么这个线程会继续等待。Object类中的notifyAll方法可以解决这个问题，它可以唤醒所有等待的线程，总有一个线程执行。

[](https://img-blog.csdnimg.cn/img_convert/32b84437e3d4b785f6d7c4f03e57d08f.png)

如上图所示，一个线程通过1号门进入Entry Set(入口区)，如果在入口区没有线程等待，那么这个线程就会获取监视器成为监视器的Owner，然后执行监视区域的代码。如果在入口区中有其它线程在等待，那么新来的线程也会和这些线程一起等待。线程在持有监视器的过程中，有两个选择，一个是正常执行监视器区域的代码，释放监视器，通过5号门退出监视器；还有可能等待某个条件的出现，于是它会通过3号门到Wait Set（等待区）休息，直到相应的条件满足后再通过4号门进入重新获取监视器再执行。

注意：

当一个线程释放监视器时，在入口区和等待区的等待线程都会去竞争监视器，如果入口区的线程赢了，会从2号门进入；如果等待区的线程赢了会从4号门进入。只有通过3号门才能进入等待区，在等待区中的线程只有通过4号门才能退出等待区，也就是说一个线程只有在持有监视器时才能执行wait操作，处于等待的线程只有再次获得监视器才能退出等待状态。

**4 锁的优化**

从JDK5引入了现代操作系统新增加的CAS原子操作（ JDK5中并没有对synchronized关键字做优化，而是体现在J.U.C中，所以在该版本concurrent包有更好的性能 ），从JDK6开始，就对synchronized的实现机制进行了较大调整，包括使用JDK5引进的CAS自旋之外，还增加了自适应的CAS自旋、锁消除、锁膨胀、偏向锁、轻量级锁这些优化策略。由于此关键字的优化使得性能极大提高，同时语义清晰、操作简单、无需手动关闭，所以推荐在允许的情况下尽量使用此关键字，同时在性能上此关键字还有优化的空间。

锁主要存在四种状态，依次是**：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态**，锁可以从偏向锁升级到轻量级锁，再升级的重量级锁。但是锁的升级是单向的，也就是说只能从低到高升级，不会出现锁的降级。

在 JDK 1.6 中默认是开启偏向锁和轻量级锁的，可以通过-XX:-UseBiasedLocking来禁用偏向锁。

**4.1 自旋锁**

线程的阻塞和唤醒需要CPU从用户态转为核心态，频繁的阻塞和唤醒对CPU来说是一件负担很重的工作，势必会给系统的并发性能带来很大的压力。同时我们发现在许多应用上面，对象锁的锁状态只会持续很短一段时间，为了这一段很短的时间频繁地阻塞和唤醒线程是非常不值得的。

所以引入自旋锁，何谓自旋锁？

所谓自旋锁，就是指当一个线程尝试获取某个锁时，如果该锁已被其他线程占用，就一直循环检测锁是否被释放，而不是进入线程挂起或睡眠状态。

自旋锁适用于锁保护的临界区很小的情况，临界区很小的话，锁占用的时间就很短。自旋等待不能替代阻塞，虽然它可以避免线程切换带来的开销，但是它占用了CPU处理器的时间。如果持有锁的线程很快就释放了锁，那么自旋的效率就非常好，反之，自旋的线程就会白白消耗掉处理的资源，它不会做任何有意义的工作，典型的占着茅坑不拉屎，这样反而会带来性能上的浪费。所以说，自旋等待的时间（自旋的次数）必须要有一个限度，如果自旋超过了定义的时间仍然没有获取到锁，则应该被挂起。

自旋锁在JDK 1.4.2中引入，默认关闭，但是可以使用-XX:+UseSpinning开开启，在JDK1.6中默认开启。同时自旋的默认次数为10次，可以通过参数-XX:PreBlockSpin来调整。

如果通过参数-XX:PreBlockSpin来调整自旋锁的自旋次数，会带来诸多不便。假如将参数调整为10，但是系统很多线程都是等你刚刚退出的时候就释放了锁（假如多自旋一两次就可以获取锁），是不是很尴尬。于是JDK1.6引入自适应的自旋锁，让虚拟机会变得越来越聪明。

**4.2 适应性自旋锁**

JDK 1.6引入了更加聪明的自旋锁，即自适应自旋锁。所谓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。那它如何进行适应性自旋呢？

**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，那么在以后要或者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。**

有了自适应自旋锁，随着程序运行和性能监控信息的不断完善，虚拟机对程序锁的状况预测会越来越准确，虚拟机会变得越来越聪明。

**4.3 锁消除**

为了保证数据的完整性，在进行操作时需要对这部分操作进行同步控制，但是在有些情况下，JVM检测到不可能存在共享数据竞争，这是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

如果不存在竞争，为什么还需要加锁呢？所以锁消除可以节省毫无意义的请求锁的时间。变量是否逃逸，对于虚拟机来说需要使用数据流分析来确定，但是对于程序员来说这还不清楚么？在明明知道不存在数据竞争的代码块前加上同步吗？但是有时候程序并不是我们所想的那样？虽然没有显示使用锁，但是在使用一些JDK的内置API时，如StringBuffer、Vector、HashTable等，这个时候会存在隐形的加锁操作。比如StringBuffer的append()方法，Vector的add()方法：

public void vectorTest(){

Vector<String> vector = new Vector<String>();

for(int i = 0 ; i < 10 ; i++){

vector.add(i + "");

}

System.out.println(vector);

}

在运行这段代码时，JVM可以明显检测到变量vector没有逃逸出方法vectorTest()之外，所以JVM可以大胆地将vector内部的加锁操作消除。

**4.4 锁膨胀**

在使用同步锁的时候，需要让同步块的作用范围尽可能小—仅在共享数据的实际作用域中才进行同步，这样做的目的是 为了使需要同步的操作数量尽可能缩小，如果存在锁竞争，那么等待锁的线程也能尽快拿到锁。

在大多数的情况下，上述观点是正确的。但是如果一系列的连续加锁解锁操作，可能会导致不必要的性能损耗，所以引入锁粗话的概念。

锁粗话概念比较好理解，就是将多个连续的加锁、解锁操作连接在一起，扩展成一个范围更大的锁

如上面实例：

**vector每次add的时候都需要加锁操作，JVM检测到对同一个对象（vector）连续加锁、解锁操作，会合并一个更大范围的加锁、解锁操作，即加锁解锁操作会移到for循环之外。**

**4.5 偏向锁**

偏向锁是JDK6中的重要引进，因为HotSpot作者经过研究实践发现，在大多数情况下，锁不仅不存在多线程竞争，而且总是由同一线程多次获得，为了让线程获得锁的代价更低，引进了偏向锁。

偏向锁是在单线程执行代码块时使用的机制，如果在多线程并发的环境下（即线程A尚未执行完同步代码块，线程B发起了申请锁的申请），则一定会转化为轻量级锁或者重量级锁。

在JDK5中偏向锁默认是关闭的，而到了JDK6中偏向锁已经默认开启。如果并发数较大同时同步代码块执行时间较长，则被多个线程同时访问的概率就很大，就可以使用参数-XX:-UseBiasedLocking来禁止偏向锁(但这是个JVM参数，不能针对某个对象锁来单独设置)。

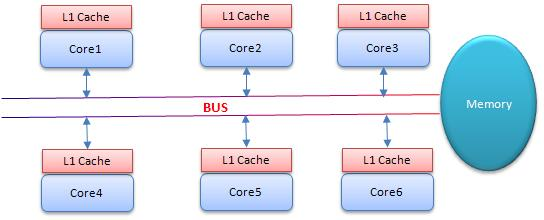
引入偏向锁主要目的是：为了在没有多线程竞争的情况下尽量减少不必要的轻量级锁执行路径。因为轻量级锁的加锁解锁操作是需要依赖多次CAS原子指令的，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令（由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗也必须小于节省下来的CAS原子指令的性能消耗）。

轻量级锁是为了在线程交替执行同步块时提高性能，而偏向锁则是在只有一个线程执行同步块时进一步提高性能。

那么偏向锁是如何来减少不必要的CAS操作呢？首先我们看下无竞争下锁存在什么问题：

**现在几乎所有的锁都是可重入的，即已经获得锁的线程可以多次锁住/解锁监视对象，按照之前的HotSpot设计，每次加锁/解锁都会涉及到一些CAS操作（比如对等待队列的CAS操作），CAS操作会延迟本地调用，因此偏向锁的想法是 一旦线程第一次获得了监视对象，之后让监视对象“偏向”这个线程，之后的多次调用则可以避免CAS操作，说白了就是置个变量，如果发现为true则无需再走各种加锁/解锁流程。**

CAS为什么会引入本地延迟？这要从SMP（对称多处理器）架构说起，下图大概表明了SMP的结构：

[](https://img-blog.csdnimg.cn/img_convert/7f26b659e5b93fdcc8de3058c9a8c049.png)

SMP（对称多处理器）架构

其意思是 所有的CPU会共享一条系统总线（BUS），靠此总线连接主存。每个核都有自己的一级缓存，各核相对于BUS对称分布，因此这种结构称为“对称多处理器”。

而CAS的全称为Compare-And-Swap，是一条CPU的原子指令，其作用是让CPU比较后原子地更新某个位置的值，经过调查发现，其实现方式是基于硬件平台的汇编指令，就是说CAS是靠硬件实现的，JVM只是封装了汇编调用，那些AtomicInteger类便是使用了这些封装后的接口。

例如：Core1和Core2可能会同时把主存中某个位置的值Load到自己的L1 Cache中，当Core1在自己的L1 Cache中修改这个位置的值时，会通过总线，使Core2中L1 Cache对应的值“失效”，而Core2一旦发现自己L1 Cache中的值失效（称为Cache命中缺失）则会通过总线从内存中加载该地址最新的值，大家通过总线的来回通信称为“Cache一致性流量”，因为总线被设计为固定的“通信能力”，如果Cache一致性流量过大，总线将成为瓶颈。而当Core1和Core2中的值再次一致时，称为“Cache一致性”，从这个层面来说，锁设计的终极目标便是减少Cache一致性流量。

而CAS恰好会导致Cache一致性流量，如果有很多线程都共享同一个对象，当某个Core CAS成功时必然会引起总线风暴，这就是所谓的本地延迟，本质上偏向锁就是为了消除CAS，降低Cache一致性流量。

*Cache一致性：*

上面提到Cache一致性，其实是有协议支持的，现在通用的协议是MESI（最早由Intel开始支持），具体参考：[http://en.wikipedia.org/wiki/MESI\_protocol。](http://en.wikipedia.org/wiki/MESI_protocol%E3%80%82)

*Cache一致性流量的例外情况：*

其实也不是所有的CAS都会导致总线风暴，这跟Cache一致性协议有关，具体参考：<http://blogs.oracle.com/dave/entry/biased_locking_in_hotspot>

*NUMA(Non Uniform Memory Access Achitecture）架构：*

与SMP对应还有非对称多处理器架构，现在主要应用在一些高端处理器上，主要特点是没有总线，没有公用主存，每个Core有自己的内存，针对这种结构此处不做讨论。

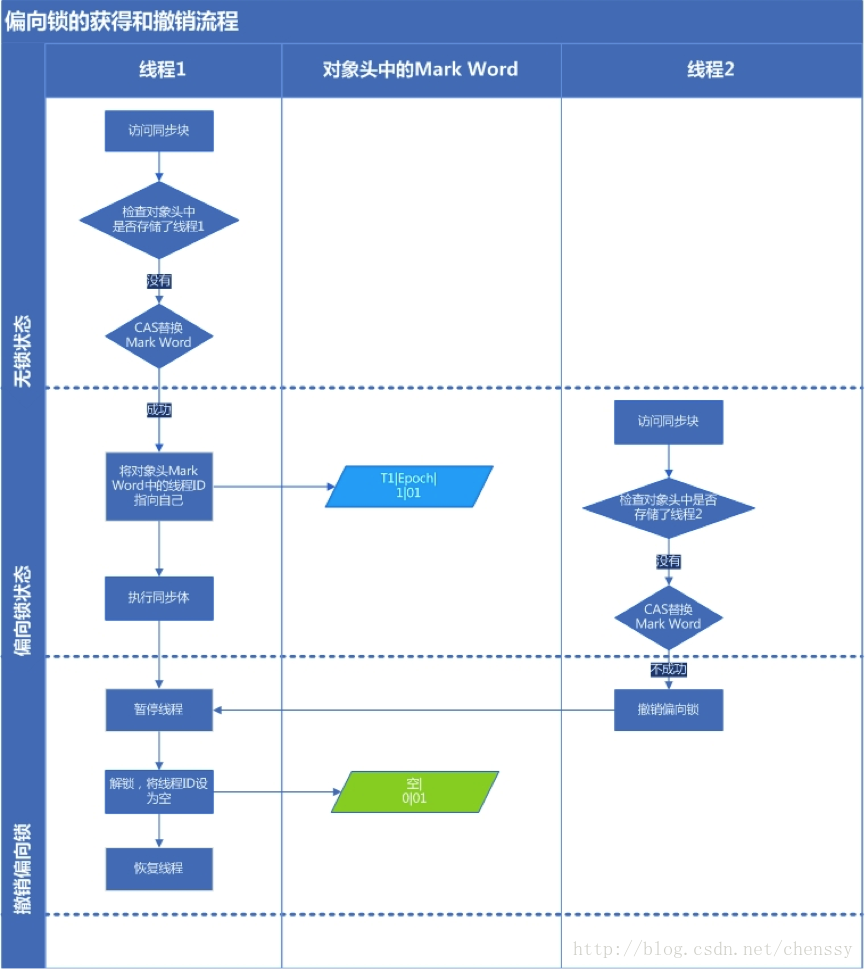
所以，当一个线程访问同步块并获取锁时，会在对象头和栈帧中的锁记录里存储锁偏向的线程ID，以后该线程进入和退出同步块时不需要花费CAS操作来争夺锁资源，只需要检查是否为偏向锁、锁标识为以及ThreadID即可，处理流程如下：

1. 检测Mark Word是否为可偏向状态，即是否为偏向锁1，锁标识位为01；
2. 若为可偏向状态，则测试线程ID是否为当前线程ID，如果是，则执行步骤（5），否则执行步骤（3）；
3. 如果测试线程ID不为当前线程ID，则通过CAS操作竞争锁，竞争成功，则将Mark Word的线程ID替换为当前线程ID，否则执行线程（4）；
4. 通过CAS竞争锁失败，证明当前存在多线程竞争情况，当到达全局安全点，获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码块；
5. 执行同步代码块；

偏向锁的释放采用了 一种只有竞争才会释放锁的机制，线程是不会主动去释放偏向锁，需要等待其他线程来竞争。偏向锁的撤销需要 等待全局安全点（这个时间点是上没有正在执行的代码）。其步骤如下：

1. 暂停拥有偏向锁的线程；
2. 判断锁对象是否还处于被锁定状态，否，则恢复到无锁状态（01），以允许其余线程竞争。是，则挂起持有锁的当前线程，并将指向当前线程的锁记录地址的指针放入对象头Mark Word，升级为轻量级锁状态（00），然后恢复持有锁的当前线程，进入轻量级锁的竞争模式；

注意：此处将 当前线程挂起再恢复的过程中并没有发生锁的转移，仍然在当前线程手中，只是穿插了个 “将对象头中的线程ID变更为指向锁记录地址的指针” 这么个事。

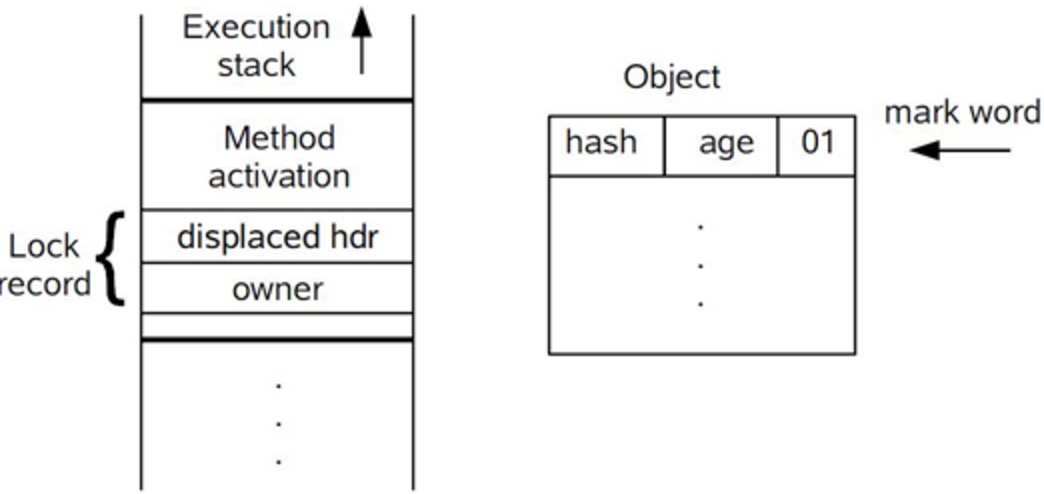
[](https://img-blog.csdnimg.cn/img_convert/43e74a861c6c3d629a9cb5f7befa7463.png)

偏向锁的获取和释放过程

**4.6 轻量级锁**

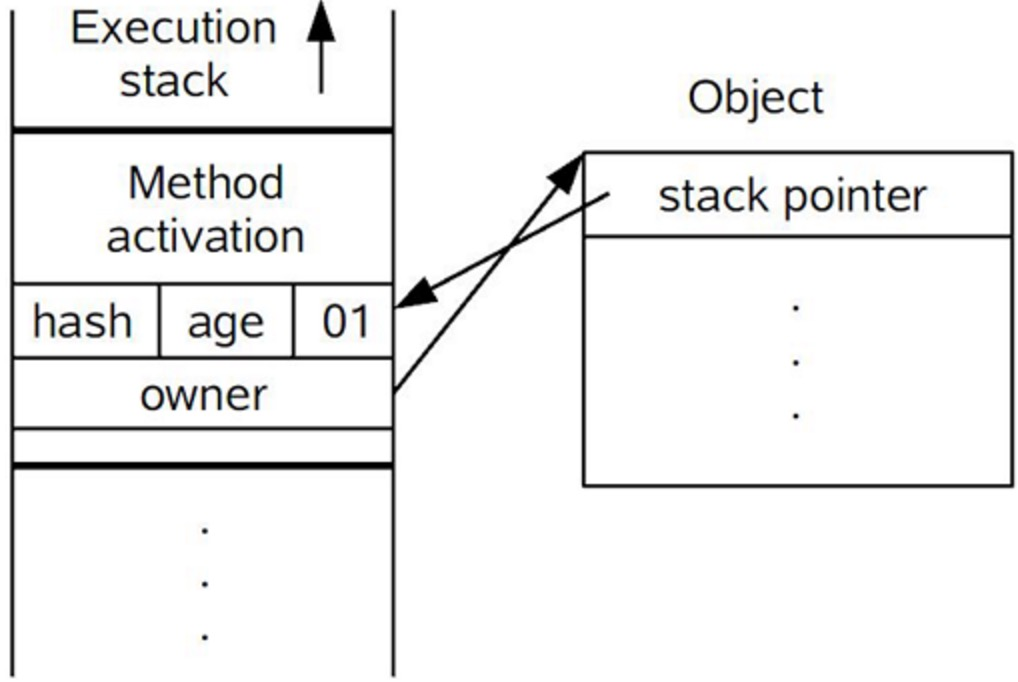
引入轻量级锁的主要目的是 在没有多线程竞争的前提下，减少传统的重量级锁使用操作系统互斥量产生的性能消耗。当关闭偏向锁功能或者多个线程竞争偏向锁导致偏向锁升级为轻量级锁，则会尝试获取轻量级锁，其步骤如下：

1. 在线程进入同步块时，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为 Displaced Mark Word。此时线程堆栈与对象头的状态如下图所示：

[](https://img-blog.csdnimg.cn/img_convert/5c9fcf877638ca602ee06e37c5da520c.png)

轻量级锁CAS操作之前线程堆栈与对象的状态

1. 拷贝对象头中的Mark Word复制到锁记录（Lock Record）中；
2. 拷贝成功后，虚拟机将使用CAS操作尝试将对象Mark Word中的Lock Word更新为指向当前线程Lock Record的指针，并将Lock record里的owner指针指向object mark word。如果更新成功，则执行步骤（4），否则执行步骤（5）；
3. 如果这个更新动作成功了，那么当前线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，即表示此对象处于轻量级锁定状态，此时线程堆栈与对象头的状态如下图所示：

[](https://img-blog.csdnimg.cn/img_convert/31c206ab7fab4f49c56b18a65f133890.png)

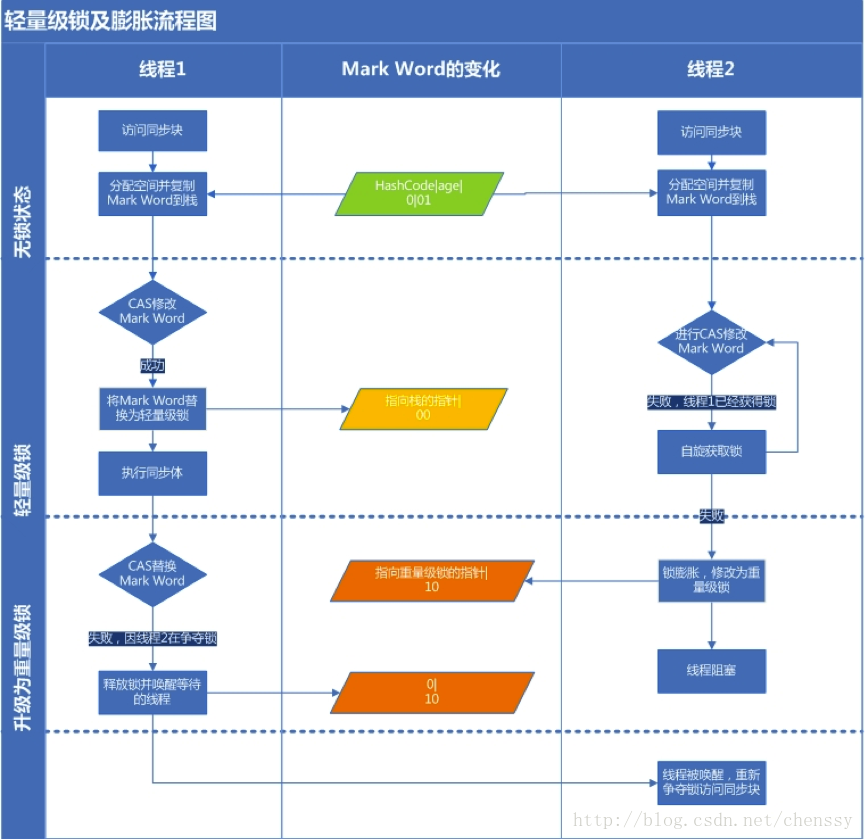
轻量级锁CAS操作之后线程堆栈与对象的状态

1. 如果这个更新操作失败了，虚拟机首先会检查对象Mark Word中的Lock Word是否指向当前线程的栈帧，如果是，就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行。否则说明多个线程竞争锁，进入自旋执行（3），若自旋结束时仍未获得锁，轻量级锁就要膨胀为重量级锁，锁标志的状态值变为“10”，Mark Word中存储的就是指向重量级锁（互斥量）的指针，当前线程以及后面等待锁的线程也要进入阻塞状态。

轻量级锁的释放也是通过CAS操作来进行的，主要步骤如下：

1. 通过CAS操作尝试把线程中复制的Displaced Mark Word对象替换当前的Mark Word；
2. 如果替换成功，整个同步过程就完成了，恢复到无锁状态（01）；
3. 如果替换失败，说明有其他线程尝试过获取该锁（此时锁已膨胀），那就要在释放锁的同时，唤醒被挂起的线程；

对于轻量级锁，其性能提升的依据是 “对于绝大部分的锁，在整个生命周期内都是不会存在竞争的”，如果打破这个依据则除了互斥的开销外，还有额外的CAS操作，因此在有多线程竞争的情况下，轻量级锁比重量级锁更慢。

[](https://img-blog.csdnimg.cn/img_convert/86a1665f6dfc8fcaef33c4fec5a115a2.png)

轻量级锁的获取和释放过程

1. 为什么升级为轻量锁时要把对象头里的Mark Word复制到线程栈的锁记录中呢？

因为在申请对象锁时 需要以该值作为CAS的比较条件，同时在升级到重量级锁的时候，能通过这个比较判定是否在持有锁的过程中此锁被其他线程申请过，如果被其他线程申请了，则在释放锁的时候要唤醒被挂起的线程。

1. 为什么会尝试CAS不成功以及什么情况下会不成功？

CAS本身是不带锁机制的，其是通过比较而来。假设如下场景：线程A和线程B都在对象头里的锁标识为无锁状态进入，那么如线程A先更新对象头为其锁记录指针成功之后，线程B再用CAS去更新，就会发现此时的对象头已经不是其操作前的对象HashCode了，所以CAS会失败。也就是说，只有两个线程并发申请锁的时候会发生CAS失败。

然后线程B进行CAS自旋，等待对象头的锁标识重新变回无锁状态或对象头内容等于对象HashCode（因为这是线程B做CAS操作前的值），这也就意味着线程A执行结束（参见后面轻量级锁的撤销，只有线程A执行完毕撤销锁了才会重置对象头），此时线程B的CAS操作终于成功了，于是线程B获得了锁以及执行同步代码的权限。如果线程A的执行时间较长，线程B经过若干次CAS时钟没有成功，则锁膨胀为重量级锁，即线程B被挂起阻塞、等待重新调度。

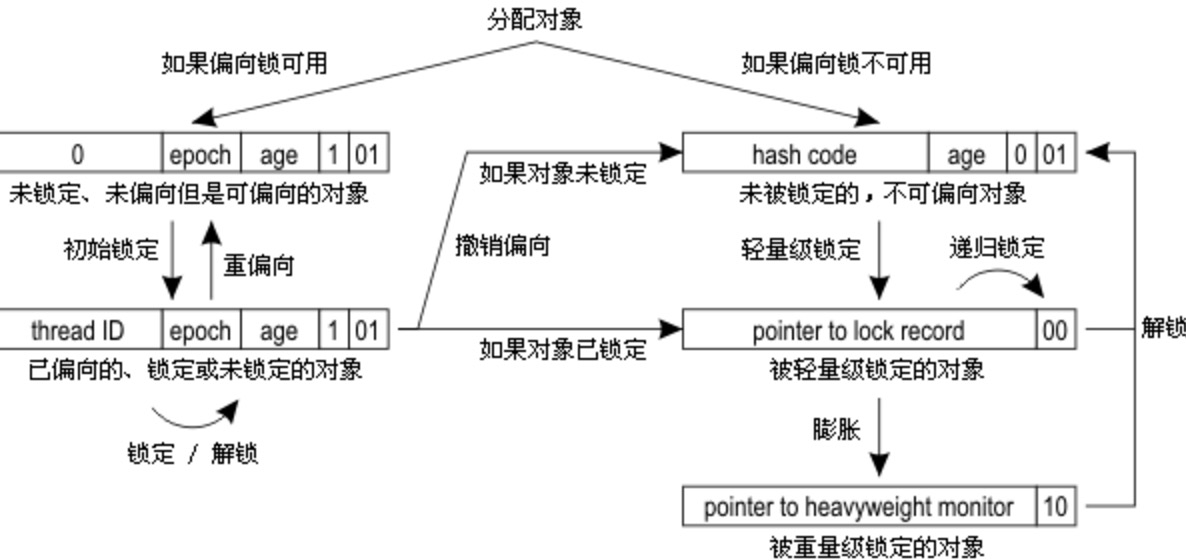
此处，如何理解“轻量级”？“轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的。但是，首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用产生的性能消耗。

轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，必然就会导致轻量级锁膨胀为重量级锁。

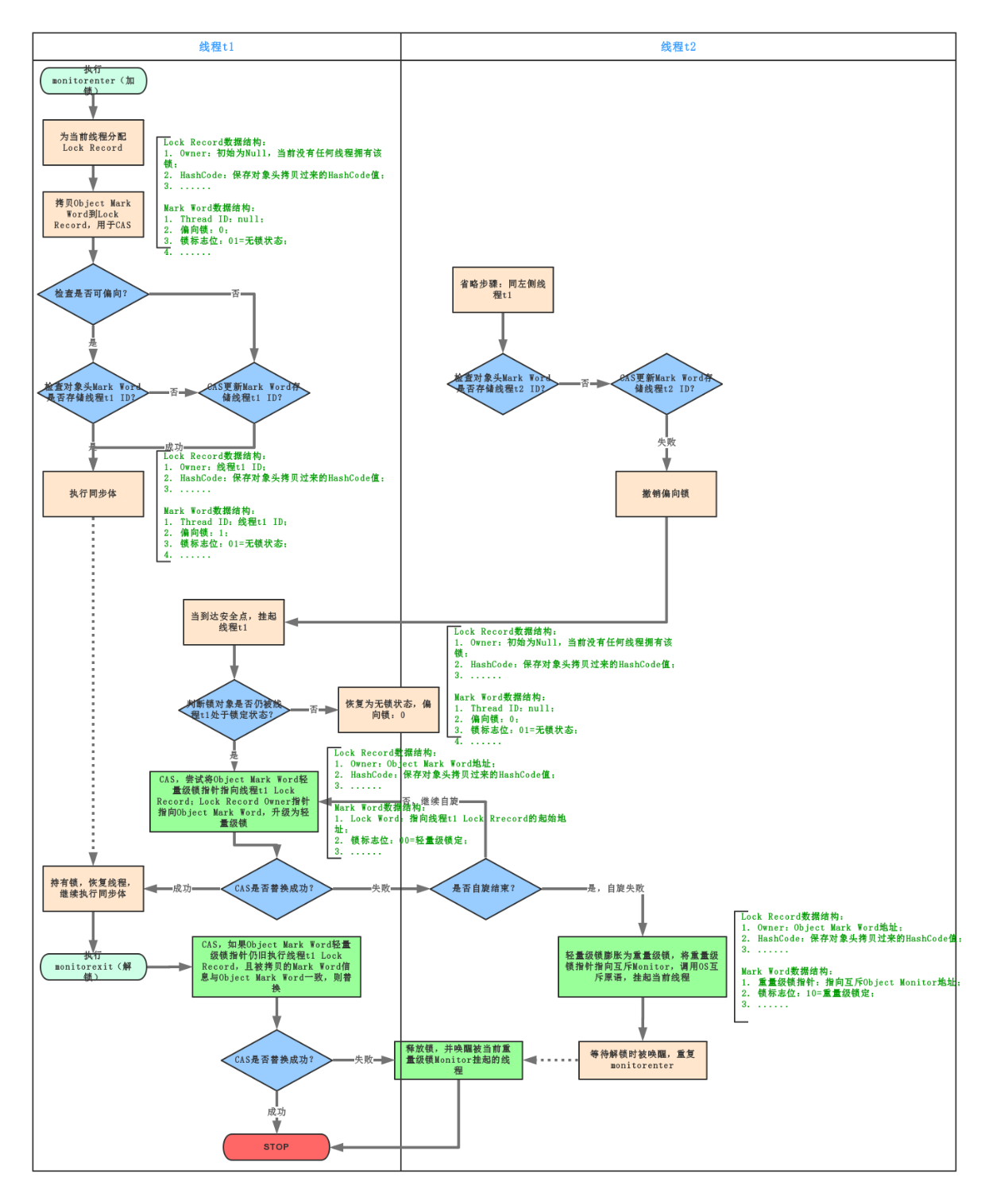
**4.7 重量级锁**

Synchronized是通过对象内部的一个叫做 监视器锁（Monitor）来实现的。但是监视器锁本质又是依赖于底层的操作系统的Mutex Lock来实现的。而操作系统实现线程之间的切换这就需要从用户态转换到核心态，这个成本非常高，状态之间的转换需要相对比较长的时间，这就是为什么Synchronized效率低的原因。因此，这种依赖于操作系统Mutex Lock所实现的锁我们称之为 “重量级锁”。

**4.8 重量级锁、轻量级锁和偏向锁之间转换**

[](https://img-blog.csdnimg.cn/img_convert/5676e39f77aa1eed694c9981df79d095.png)

**重量级锁、轻量级锁和偏向锁之间转换**

[](https://img-blog.csdnimg.cn/img_convert/6c8febd362e56d6573991b5bf4eeed31.png)

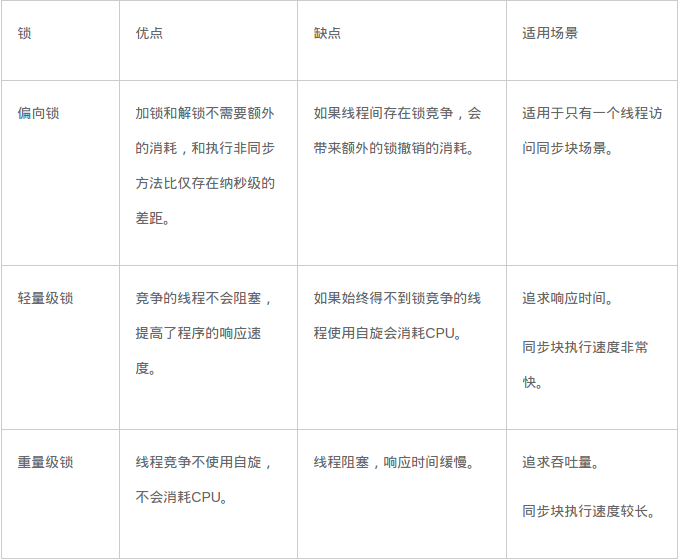
Synchronized偏向锁、轻量级锁及重量级锁转换流程

**5 锁的优劣**

各种锁并不是相互代替的，而是在不同场景下的不同选择，绝对不是说重量级锁就是不合适的。每种锁是只能升级，不能降级，即由偏向锁->轻量级锁->重量级锁，而这个过程就是开销逐渐加大的过程。

1. 如果是单线程使用，那偏向锁毫无疑问代价最小，并且它就能解决问题，连CAS都不用做，仅仅在内存中比较下对象头就可以了；
2. 如果出现了其他线程竞争，则偏向锁就会升级为轻量级锁；
3. 如果其他线程通过一定次数的CAS尝试没有成功，则进入重量级锁；

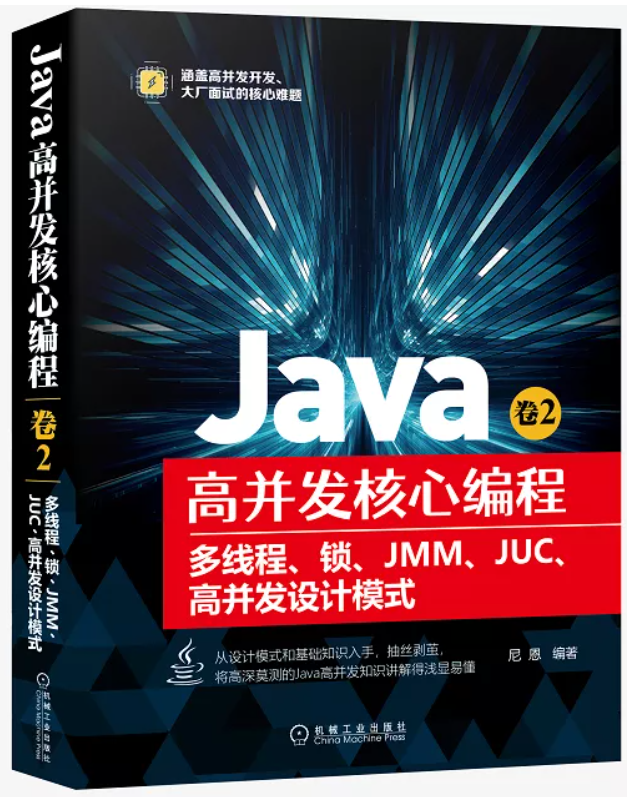
在第3种情况下进入同步代码块就 要做偏向锁建立、偏向锁撤销、轻量级锁建立、升级到重量级锁，最终还是得靠重量级锁来解决问题，那这样的代价就比直接用重量级锁要大不少了。所以使用哪种技术，一定要看其所处的环境及场景，在绝大多数的情况下，偏向锁是有效的，这是基于HotSpot作者发现的“大多数锁只会由同一线程并发申请”的经验规律。

[](https://img-blog.csdnimg.cn/img_convert/0894ed62c588d7ad6dbd4640d407e356.png)

**6.Synchronized原理的推荐学习书籍**

Synchronized原理非常复杂，如果上面的问题不清楚，请前往阅读：

《Java高并发核心编程（卷2）：多线程、锁、JMM、JUC、高并发设计模式》。

[](https://img-blog.csdnimg.cn/20210413202952845.png)

**五：线程池面试题**

**1、 什么是线程池？ 为什么要使用它？**

创建线程要花费昂贵的资源和时间，如果任务来了才创建线程那么响应时间会变长，而且一个进程能创建的线程数有限。

多线程技术主要解决处理器单元内多个线程执行的问题，它可以显著减少处理器单元的闲置时间，增加处理器单元的吞吐能力。  
假设一个服务器完成一项任务所需时间为：T1 创建线程时间，T2 在线程中执行任务的时间，T3 销毁线程时间。

如果：T1 + T3 远大于 T2，则可以采用线程池，以提高服务器性能。

一个线程池包括以下四个基本组成部分：  
1、线程池管理器（ThreadPool）：用于创建并管理线程池，包括 创建线程池，销毁线程池，添加新任务；  
2、工作线程（PoolWorker）：线程池中线程，在没有任务时处于等待状态，可以循环的执行任务；  
3、任务接口（Task）：每个任务必须实现的接口，以供工作线程调度任务的执行，它主要规定了任务的入口，任务执行完后的收尾工作，任务的执行状态等；  
4、任务队列（taskQueue）：用于存放没有处理的任务。提供一种缓冲机制。

线程池技术正是关注如何缩短或调整T1,T3时间的技术，从而提高服务器程序性能的。它把T1，T3分别安排在服务器程序的启动和结束的时间段或者一些空闲的时间段，这样在服务器程序处理客户请求时，不会有T1，T3的开销了。  
线程池不仅调整T1,T3产生的时间段，而且它还显著减少了创建线程的数目，看一个例子：  
假设一个服务器一天要处理50000个请求，并且每个请求需要一个单独的线程完成。在线程池中，线程数一般是固定的，所以产生线程总数不会超过线程池中线程的数目，而如果服务器不利用线程池来处理这些请求则线程总数为50000。一般线程池大小是远小于50000。所以利用线程池的服务器程序不会为了创建50000而在处理请求时浪费时间，从而提高效率。

为了避免这些问题，在程序启动的时候就创建若干线程来响应处理，它们被称为线程池，里面的线程叫工作线程。从JDK1.5开始，Java API提供了Executor框架让你可以创建不同的线程池。比如单线程池，每次处理一个任务；数目固定的线程池或者是缓存线程池（一个适合很多生存期短的任务的程序的可扩展线程池）。

**2） 什么是Executor框架？**

Executor框架同java.util.concurrent.Executor 接口在Java 5中被引入。Executor框架是一个根据一组执行策略调用，调度，执行和控制的异步任务的框架。

无限制的创建线程会引起应用程序内存溢出。所以创建一个线程池是个更好的的解决方案，因为可以限制线程的数量并且可以回收再利用这些线程。利用Executor框架可以非常方便的创建一个线程池。

60） Executors类是什么？

Executors为Executor，ExecutorService，ScheduledExecutorService，ThreadFactory和Callable类提供了一些工具方法。

Executors可以用于方便的创建线程池。

**3） 什么是阻塞队列？如何使用阻塞队列来实现生产者-消费者模型？**

java.util.concurrent.BlockingQueue的特性是：当队列是空的时，从队列中获取或删除元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。

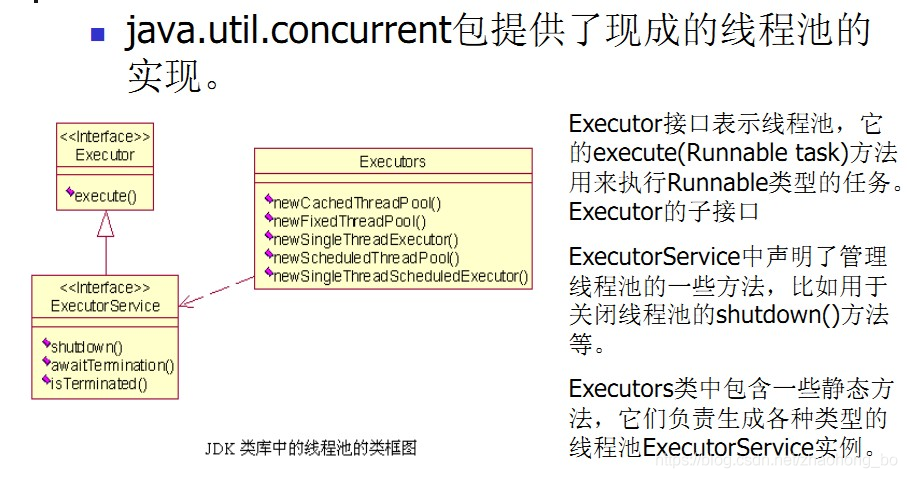
阻塞队列不接受空值，当你尝试向队列中添加空值的时候，它会抛出NullPointerException。

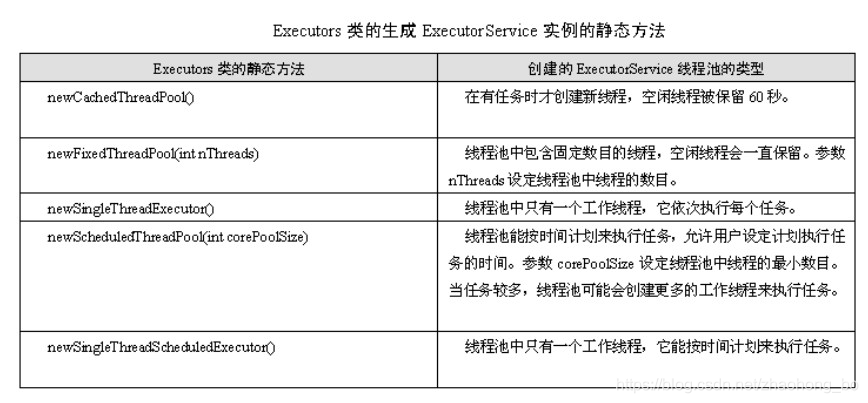
阻塞队列的实现都是线程安全的，所有的查询方法都是原子的并且使用了内部锁或者其他形式的并发控制。

BlockingQueue 接口是java collections框架的一部分，它主要用于实现生产者-消费者问题。

**4.常见线程池的快捷创建方式**

①**newSingleThreadExecutor**  
单个线程的线程池，即线程池中每次只有一个线程工作，单线程串行执行任务  
②**newFixedThreadExecutor**(n)  
固定数量的线程池，没提交一个任务就是一个线程，直到达到线程池的最大数量，然后后面进入等待队列直到前面的任务完成才继续执行  
③**newCacheThreadExecutor**（推荐使用）  
可缓存线程池，当线程池大小超过了处理任务所需的线程，那么就会回收部分空闲（一般是60秒无执行）的线程，当有任务来时，又智能的添加新线程来执行。  
④**newScheduleThreadExecutor**  
大小无限制的线程池，支持定时和周期性的执行线程

java提供的线程池更加强大，相信理解线程池的工作原理，看类库中的线程池就不会感到陌生了。  
[](https://img-blog.csdnimg.cn/20190414223213863.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3poYW9ob25nX2Jv,size_16,color_FFFFFF,t_70)

[](https://img-blog.csdnimg.cn/20190414223234513.png?x-oss-process=image/watermark,type_ZmFuZ3poZW5naGVpdGk,shadow_10,text_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3poYW9ob25nX2Jv,size_16,color_FFFFFF,t_70)

要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，很有可能配置的线程池不是较优的，因此在Executors类里面提供了一些静态工厂，生成一些常用的线程池。

**4.1 newSingleThreadExecutor**

创建一个单线程的线程池。这个线程池只有一个线程在工作，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。

**4.2 newFixedThreadPool**

创建固定大小的线程池。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。

**4.3 newCachedThreadPool**

创建一个可缓存的线程池。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）的线程，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。此线程池不会对线程池大小做限制，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。

**4.4 newScheduledThreadPool**

创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。

**5、为什么不建议使用 Executors静态工厂构建线程池**

阿里巴巴Java开发手册，明确指出**不允许**使用Executors静态工厂构建线程池  
原因如下：  
线程池不允许使用Executors去创建，而是通过ThreadPoolExecutor的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

说明：Executors返回的线程池对象的弊端如下：

1：FixedThreadPool 和 SingleThreadPool：  
允许的请求队列（底层实现是LinkedBlockingQueue）长度为Integer.MAX\_VALUE，可能会堆积大量的请求，从而导致OOM  
2：CachedThreadPool 和 ScheduledThreadPool  
允许的创建线程数量为Integer.MAX\_VALUE，可能会创建大量的线程，从而导致OOM。  
[](https://img-blog.csdnimg.cn/20190415205333452.png)  
创建线程池的正确姿势

避免使用Executors创建线程池，主要是避免使用其中的默认实现，那么我们可以自己直接调用ThreadPoolExecutor的构造函数来自己创建线程池。在创建的同时，给BlockQueue指定容量就可以了。

private static ExecutorService executor = new ThreadPoolExecutor(10, 10,

60L, TimeUnit.SECONDS,

new ArrayBlockingQueue(10));

或者是使用开源类库：开源类库，如apache和guava等。

**6、线程池常用参数**

*/\*\**

*\* Creates a new {@code ThreadPoolExecutor} with the given initial*

*\* parameters.*

*\**

*\* @param corePoolSize the number of threads to keep in the pool, even*

*\* if they are idle, unless {@code allowCoreThreadTimeOut} is set*

*\* @param maximumPoolSize the maximum number of threads to allow in the*

*\* pool*

*\* @param keepAliveTime when the number of threads is greater than*

*\* the core, this is the maximum time that excess idle threads*

*\* will wait for new tasks before terminating.*

*\* @param unit the time unit for the {@code keepAliveTime} argument*

*\* @param workQueue the queue to use for holding tasks before they are*

*\* executed. This queue will hold only the {@code Runnable}*

*\* tasks submitted by the {@code execute} method.*

*\* @param threadFactory the factory to use when the executor*

*\* creates a new thread*

*\* @param handler the handler to use when execution is blocked*

*\* because the thread bounds and queue capacities are reached*

*\* @throws IllegalArgumentException if one of the following holds:<br>*

*\* {@code corePoolSize < 0}<br>*

*\* {@code keepAliveTime < 0}<br>*

*\* {@code maximumPoolSize <= 0}<br>*

*\* {@code maximumPoolSize < corePoolSize}*

*\* @throws NullPointerException if {@code workQueue}*

*\* or {@code threadFactory} or {@code handler} is null*

*\*/*

public ThreadPoolExecutor(int corePoolSize,

int maximumPoolSize,

long keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue,

ThreadFactory threadFactory,

RejectedExecutionHandler handler) { }

corePoolSize：核心线程数量，会一直存在，除非allowCoreThreadTimeOut设置为true  
maximumPoolSize：线程池允许的最大线程池数量  
keepAliveTime：线程数量超过corePoolSize，空闲线程的最大超时时间  
unit：超时时间的单位  
workQueue：工作队列，保存未执行的Runnable 任务  
threadFactory：创建线程的工厂类  
handler：当线程已满，工作队列也满了的时候，会被调用。被用来实现各种拒绝策略。

**7、线程池的关闭**

关闭线程池可以调用shutdownNow和shutdown两个方法来实现

shutdownNow：对正在执行的任务全部发出interrupt()，停止执行，对还未开始执行的任务全部取消，并且返回还没开始的任务列表。

shutdown：当我们调用shutdown后，线程池将不再接受新的任务，但也不会去强制终止已经提交或者正在执行中的任务。

**8、初始化线程池时线程数的选择**

如果任务是IO密集型，一般线程数需要设置2倍CPU数以上，以此来尽量利用CPU资源。

如果任务是CPU密集型，一般线程数量只需要设置CPU数加1即可，更多的线程数也只能增加上下文切换，不能增加CPU利用率。

如果任务是混合型，有一个公式，具体请参考：《Java高并发核心编程（卷2）》

**9、线程池都有哪几种工作队列**

1、ArrayBlockingQueue

是一个基于数组结构的有界阻塞队列，此队列按 FIFO（先进先出）原则对元素进行排序。

2、LinkedBlockingQueue

一个基于链表结构的阻塞队列，此队列按FIFO （先进先出） 排序元素，吞吐量通常要高于ArrayBlockingQueue。静态工厂方法Executors.newFixedThreadPool()使用了这个队列

3、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LinkedBlockingQueue，静态工厂方法Executors.newCachedThreadPool使用了这个队列。

4、PriorityBlockingQueue

一个具有优先级的无限阻塞队列。

**六、threadlocal面试题**

**1) 什么是ThreadLocal变量？**

ThreadLocal是Java里一种特殊的变量。每个线程都有一个ThreadLocal就是每个线程都拥有了自己独立的一个变量，竞争条件被彻底消除了。如果为每个线程提供一个自己独有的变量拷贝，将大大提高效率。首先，通过复用减少了代价高昂的对象的创建个数。其次，你在没有使用高代价的同步或者不变性的情况下获得了线程安全。

**2) ThreadLocal用在什么地方？**

讨论ThreadLocal用在什么地方前，我们先明确下，如果仅仅就一个线程，那么都不用谈ThreadLocal的，**ThreadLocal是用在多线程的场景的！！！**

ThreadLocal归纳下来就2类用途：

* **保存线程上下文信息，在任意需要的地方可以获取！！！**
* **线程安全的，避免某些情况需要考虑线程安全必须同步带来的性能损失！！！**

**保存线程上下文信息，在任意需要的地方可以获取！！！**

由于ThreadLocal的特性，同一线程在某地方进行设置，在随后的任意地方都可以获取到。从而可以用来保存线程上下文信息。

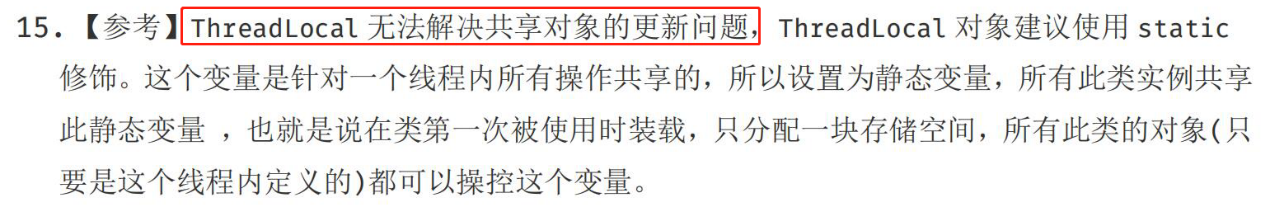
常用的比如每个请求怎么把一串后续关联起来，就可以用ThreadLocal进行set，在后续的任意需要记录日志的方法里面进行get获取到请求id，从而把整个请求串起来。

还有比如Spring的事务管理，用ThreadLocal存储Connection，从而各个DAO可以获取同一Connection，可以进行事务回滚，提交等操作。

**备注：** ThreadLocal的这种用处，很多时候是用在一些优秀的框架里面的，一般我们很少接触，反而下面的场景我们接触的更多一些！

**线程安全的，避免某些情况需要考虑线程安全必须同步带来的性能损失！！！**

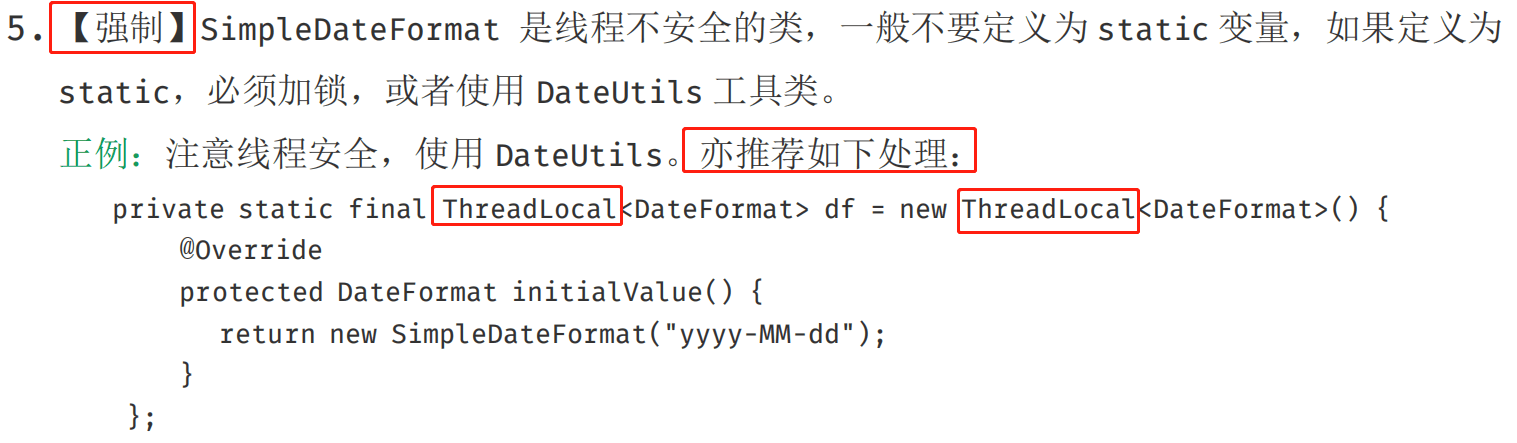
ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。但是ThreadLocal也有局限性，我们来看看阿里规范：

[](https://img-blog.csdnimg.cn/img_convert/3e2c2a81723747765bd863c3558b6237.png)

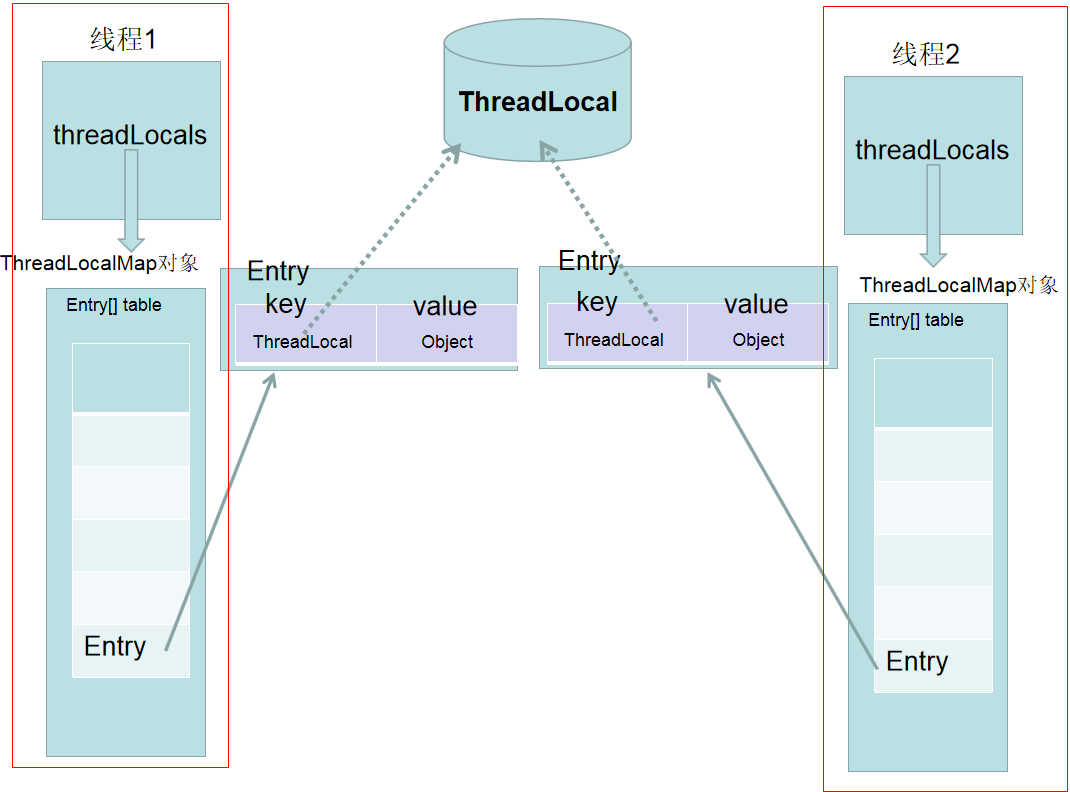
每个线程往ThreadLocal中读写数据是线程隔离，互相之间不会影响的，所以**ThreadLocal无法解决共享对象的更新问题！**

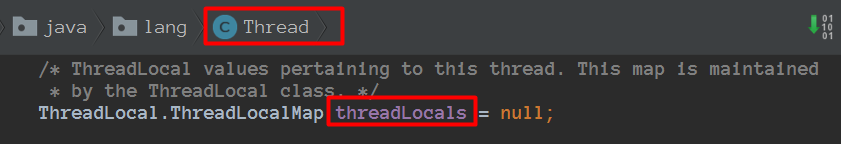
由于不需要共享信息，自然就不存在竞争问题了，从而保证了某些情况下线程的安全，以及避免了某些情况需要考虑线程安全必须同步带来的性能损失！！！

**这类场景阿里规范里面也提到了：**

[](https://img-blog.csdnimg.cn/img_convert/d5bf83bed160ee63ba58b513839c6573.png)

**3、Thread、ThreadLocalMap、ThreadLocal的关系**

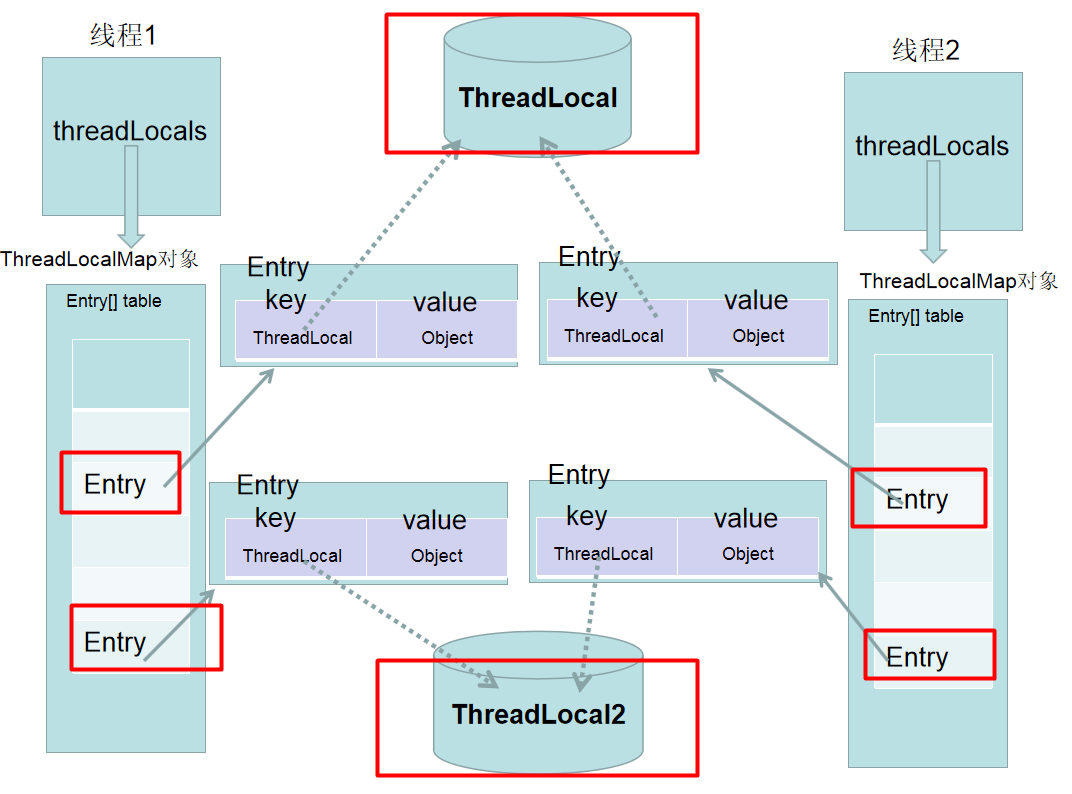
[](https://img-blog.csdnimg.cn/img_convert/89fa81fb1031aa6e41082332aa4580fd.png)

[](https://img-blog.csdnimg.cn/img_convert/cefc62d614834b5de4e15e088c0c33e2.png)

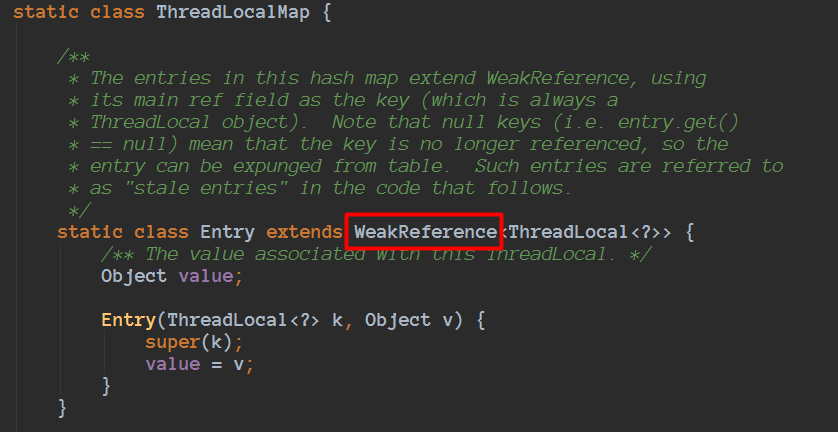
Thread类有属性变量threadLocals （类型是ThreadLocal.ThreadLocalMap），也就是说每个线程有一个自己的ThreadLocalMap ，所以每个线程往这个ThreadLocal中读写隔离的，并且是互相不会影响的。

**一个ThreadLocal只能存储一个Object对象，如果需要存储多个Object对象那么就需要多个ThreadLocal！！！**

如图：

[](https://img-blog.csdnimg.cn/img_convert/0664df7c14549edc2da804dd78cde6d7.png)

看到上面的几个图，大概思路应该都清晰了，我们Entry的key指向ThreadLocal用**虚线**表示弱引用 ，下面我们来看看ThreadLocalMap:

[](https://img-blog.csdnimg.cn/img_convert/535d48a0bf3c4ac7f412c13605702567.png)

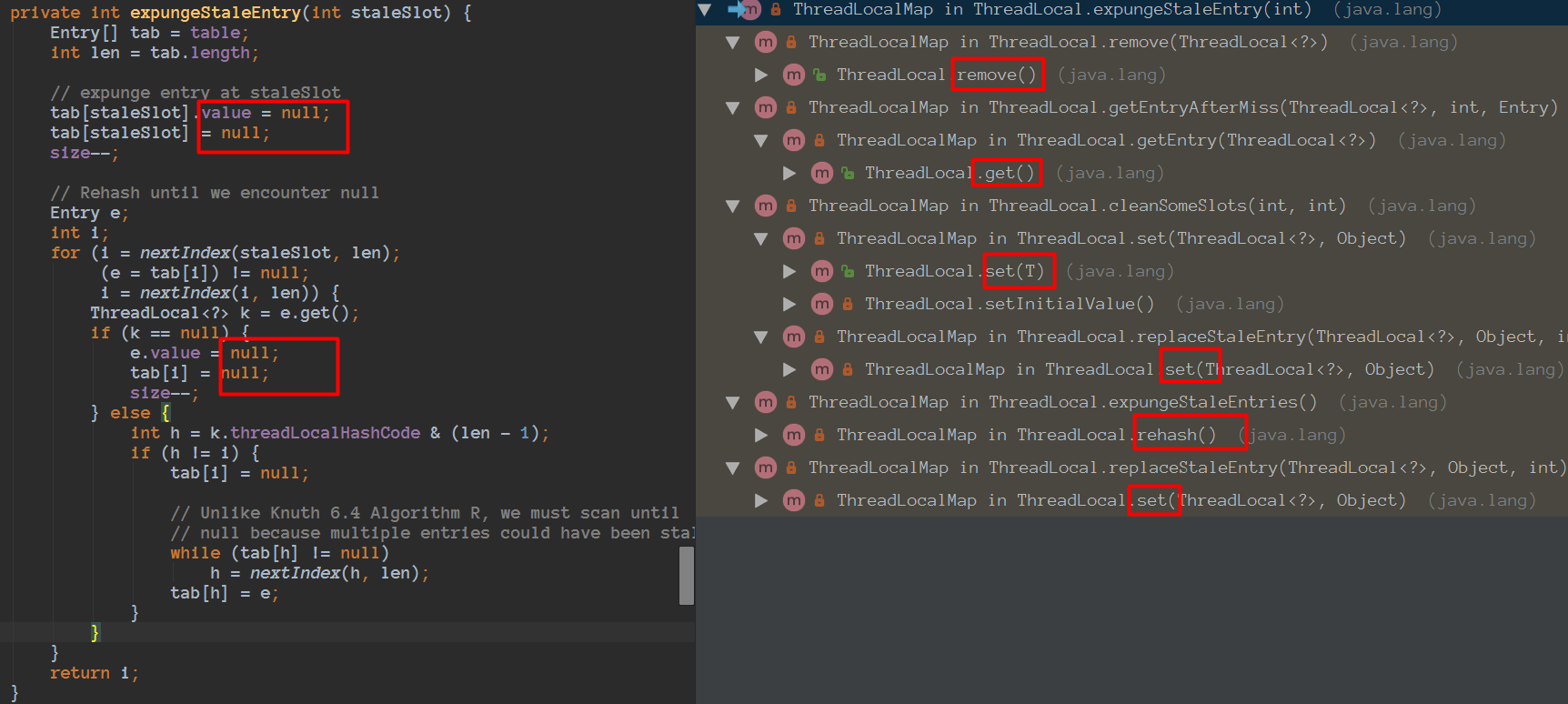
java对象的引用包括 ： 强引用，软引用，弱引用，虚引用 。

因为这里涉及到弱引用，简单说明下：

弱引用也是用来描述非必需对象的，当JVM进行垃圾回收时，无论内存是否充足，**该对象仅仅被弱引用关联**，那么就会被回收。

**当仅仅只有ThreadLocalMap中的Entry的key指向ThreadLocal的时候，ThreadLocal会进行回收的！！！**

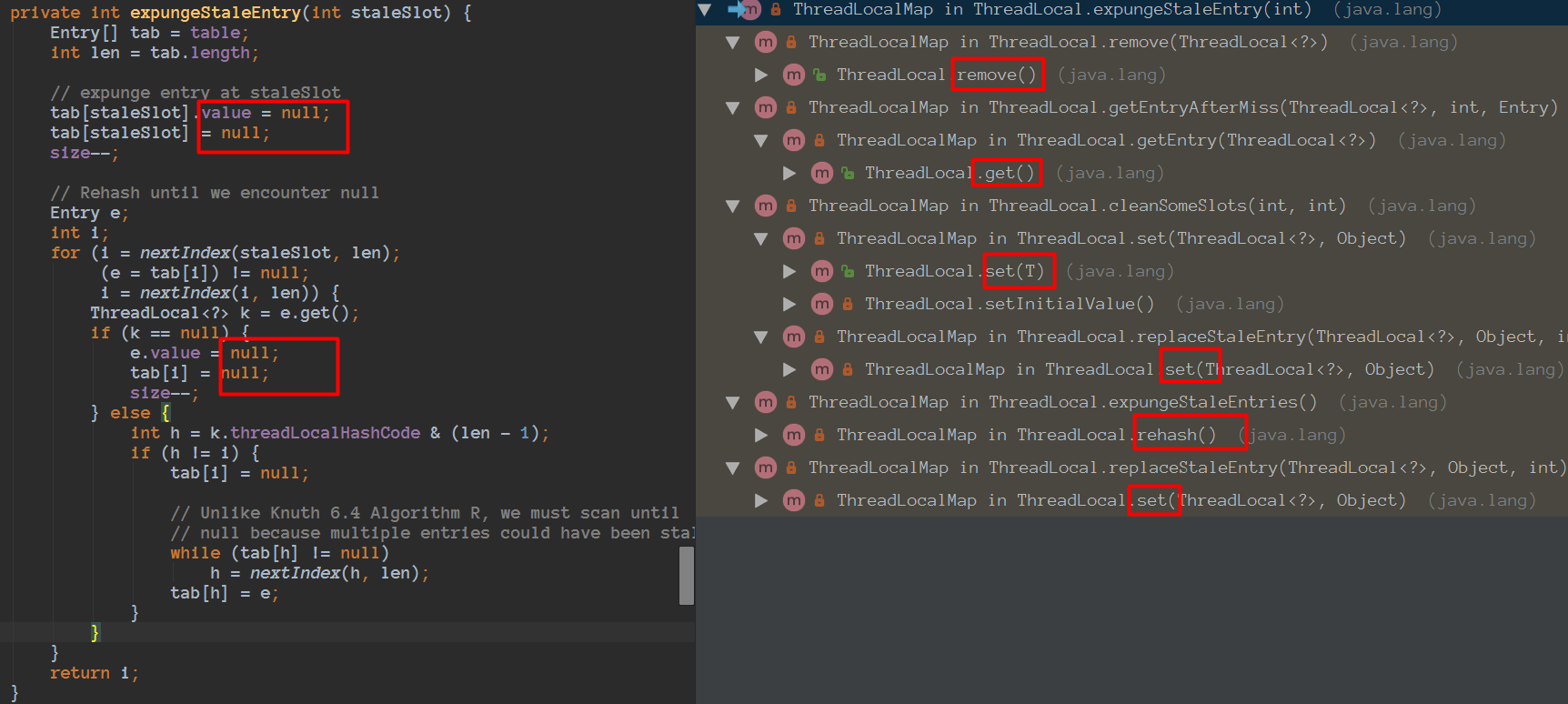
ThreadLocal被垃圾回收后，在ThreadLocalMap里对应的Entry的键值会变成null，但是Entry是强引用，那么Entry里面存储的Object，并没有办法进行回收，所以ThreadLocalMap 做了一些额外的回收工作。

[](https://img-blog.csdnimg.cn/img_convert/5db132c1a35a61320fffcb805fe13133.png)

虽然做了但是也会存在内存泄漏风险（我没有遇到过，网上很多类似场景，**所以会提到后面的ThreadLocal最佳实践！！！**）

**4、为什么要调用remove方法进行清理?**

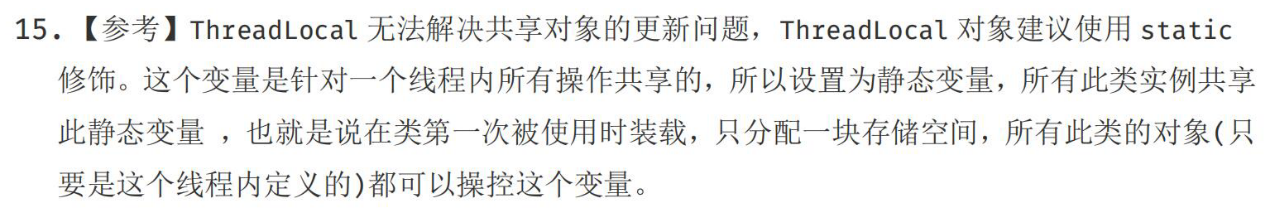
ThreadLocal被垃圾回收后，在ThreadLocalMap里对应的Entry的键值会变成null，但是Entry是强引用，那么Entry里面存储的Object，并没有办法进行回收，所以ThreadLocalMap 做了一些额外的回收工作。

[](https://img-blog.csdnimg.cn/img_convert/5db132c1a35a61320fffcb805fe13133.png)

**备注：** 很多时候，我们都是用在线程池的场景，程序不停止，线程基本不会销毁！！！

由于线程的生命周期很长，如果我们往ThreadLocal里面set了很大很大的Object对象，虽然set、get等等方法在特定的条件会调用进行额外的清理，但是**ThreadLocal被垃圾回收后，在ThreadLocalMap里对应的Entry的键值会变成null，但是后续在也没有操作set、get等方法了。**

**所以最佳实践，应该在我们不使用的时候，主动调用remove方法进行清理。**

[](https://img-blog.csdnimg.cn/img_convert/991adfb4bcbb6bda891971ea7dcc03ba.png)

这里把ThreadLocal定义为static还有一个好处就是，由于ThreadLocal有强引用在，那么在ThreadLocalMap里对应的Entry的键会永远存在，那么执行remove的时候就可以正确进行定位到并且删除！！！

**最佳实践做法应该为：**

try {

*// 其它业务逻辑*

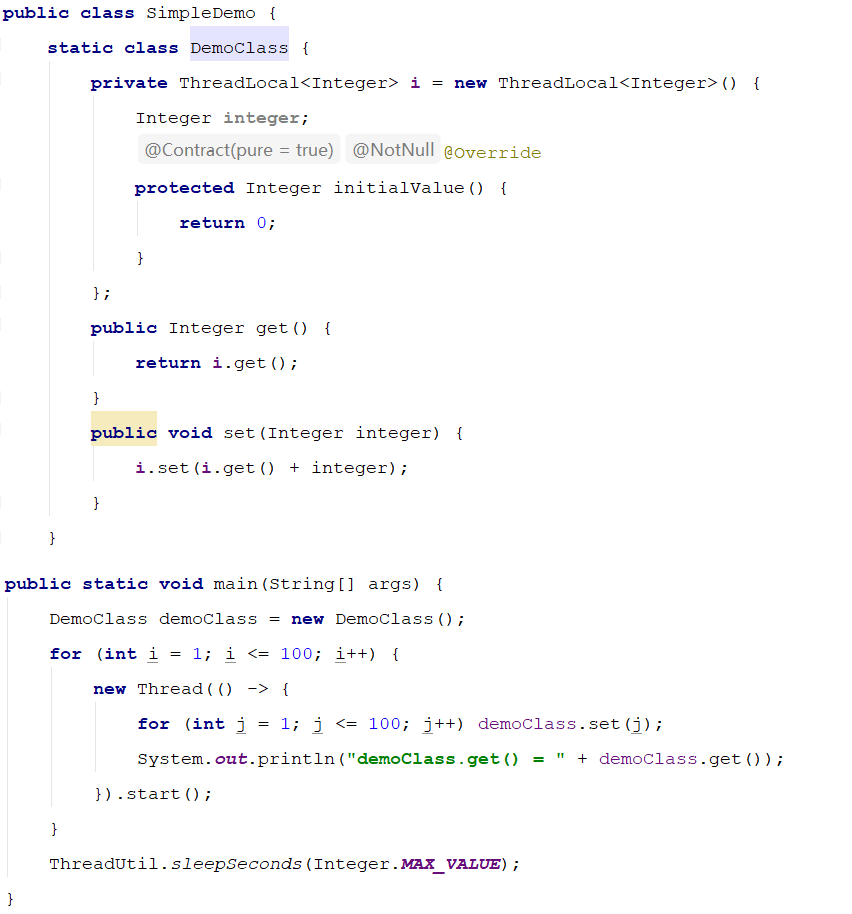
} finally {

threadLocal对象.remove();

}

[](https://img-blog.csdnimg.cn/img_convert/79dec65c47ede789aa60d9c27a34a28f.png)

**5、下面的例子，输出结果是？**

[](https://img-blog.csdnimg.cn/20210413215725866.png)

答案：100个5050