1. 什么是多线程并发和并行？

并行：是两个任务同时运行，就是甲任务进行的同时，乙任务也在进行。(需要多核CPU)

比如我跟两个网友聊天，左手操作一个电脑跟甲聊，同时右手用另一台电脑跟乙聊天，这就叫并行。

并发：指两个任务都请求运行，而处理器只能按受一个任务，就把这两个任务安排轮流进行，由于时间间隔较短，使人感觉两个任务都在运行。（表面看是CPU在同时执行多个任务，其实实际上是因为CPU瞬间切换到其他任务的速度特别快，在不同的任务之间一直在不停的切换，给不同的任务分配了不同的时间。）

如果用一台电脑我先给甲发个消息，然后立刻再给乙发消息，然后再跟甲聊，再跟乙聊。这就叫并发。

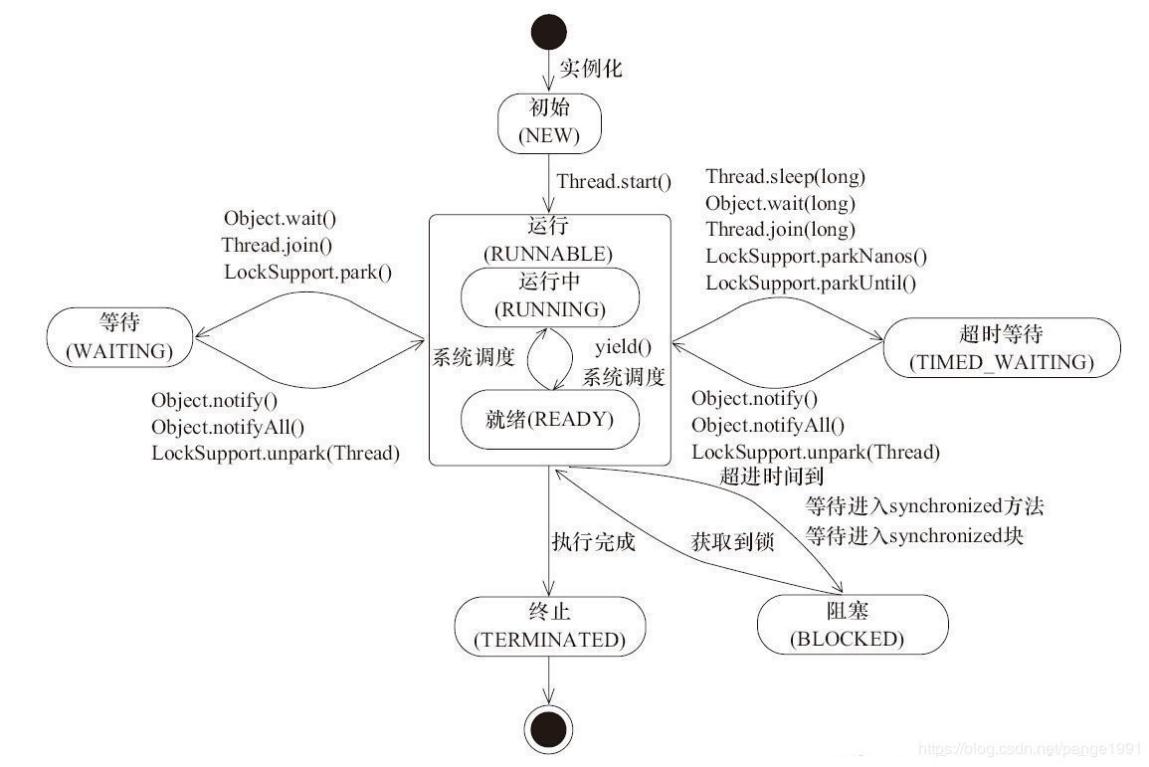
1. 什么是临界区

临界区用来表示一种公共资源或者说公共数据，可以被多个线程适用。但同时只能有一个线程适用它，一旦临界区资源被占用，其他线程要想使用它只能等待。

1. 线程的几种状态

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”。
3. 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）。
4. 阻塞(BLOCKED)：表示线程阻塞于锁。
5. 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）。
6. 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。
7. 终止(TERMINATED)：表示该线程已经执行完毕。



1. 死锁、饥饿、活锁

死锁：是指多个进程在运行过程中因争夺资源而造成的一种僵局，当进程处于这种僵持状态时，若无外力作用，它们都将无法再向前推进。

饥饿：是指一个或多个线程无法获得所需要资源，导致无法执行下去。比如它的优先级较低，而高优先级的线程不断抢占它的资源。

活锁：线程秉着谦让的原则，主动将资源释放给别的线程使用，那么就会出现资源不断在两个线程中跳动，而没有一个线程可以同时拿到所有的资源而正常执行。

1. 并发级别

由于临界区的存在，多线程之间的并发必须受到控制。根据控制并发的策略，我们可以吧并发级别进行分类，阻塞、无饥饿、无障碍、无锁、无等待。

1. JMM（java内存模型）

原子性：是指一个操作是不可中断的，即使多个线程一起执行，一旦一个操作开启就不会被其他线程干扰。对于32位系统，int型读写是原子性的，而long的读写不是原子性的。

可见性：一旦一个线程修改了某一个共享变量的值，其他线程是否立即知道这个修改。

有序性：有序性的问题是因为程序在执行时可能进行指令重排，重排后的指令顺序未必相同。

1. 线程中断

线程中有三个关于中断的方法

Public void Thread.interrupt() //中断线程，设置标志位

Public boolean Thread.isInterrupted()//判断是否中断

Public static boolean Thread.interrupted()//判断是否中断，并清除当前中断标志位

Thread.interrupt()只是设置了中断标志位，必须在线程的run()方法调用Thread.isInterrupted()让线程执行完run()方法。

Thread.sleep()方法会让当前线程休眠若干时间，若此时执行interrupt()方法，那么会抛出InterruptedException中断异常，并且删除标志位。

1. Wait()和notify()

Wait()和notify()是object类的方法，线程a调用了object.wait()就会进入等待该object的队列，必须在b线程调用object.notify()方法后才能唤醒a，notifyAll()会唤醒所有等待的线程。

wait()方法只能在synchronzied语句中执行。

1. Suspend和resume

线程suspend()方法后会挂起并不会去释放任何锁资源，直到有线程去调用该线程的resume()方法。

1. join和yield

join()等待线程执行完

yield()是当前线程让出CPU

1. Synchronized（i）

Integer是不变对象，因此当执行i++时，对i进行加锁无法实现同步，因为i++本质上是新建一个新的对象出来，用synchronized修饰都是在不同的对象上加锁

1. 信号量semaphore

信号量为多线程提供了更为强大的控制方法，广义上说信号量对锁的扩展。无论内部锁synchronized还是重入锁ReentrantLock，一次都只允许一个线程访问一个资源，而信号量却可以指定多个线程。

Public void acquire()

Public void acqurireUninterruptibly()

Public boolean tryAcquire()

Public boolean tryAquire(long timeout,TimeUnit unit)

Public void release()

// 创建一个计数阈值为5的信号量对象

// 只能5个线程同时访问

Semaphore semp = new Semaphore(5);

try {

// 申请许可

semp.acquire();

try {

// 业务逻辑

} catch (Exception e) {

} finally {

// 释放许可

semp.release();

}

} catch (InterruptedException e) {

}

1. LockSupport

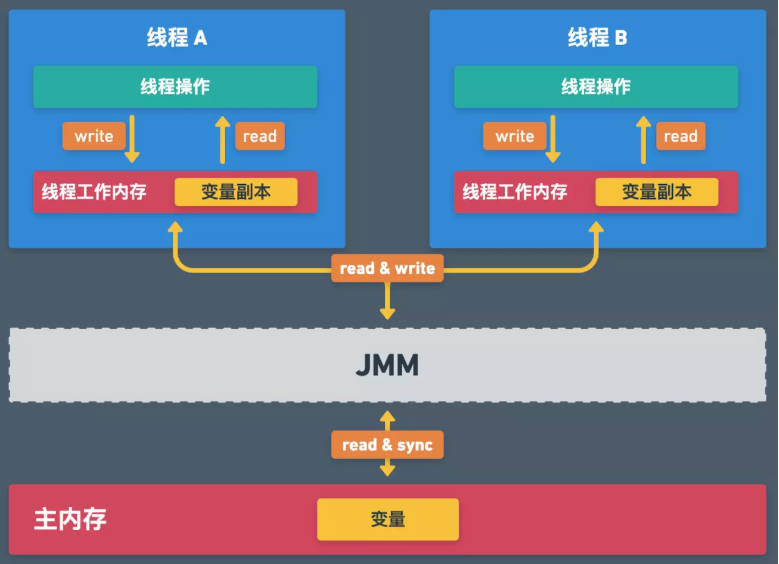
LockSupport是一个方便实用的阻塞工具类，它可以在线程内任意位置让线程阻塞。和Thread.supend()相比它弥补了由于resume()在前发生导致线程无法执行下去的情况。相比Object.wait()它不需要先获得某一个人对象的锁，也不会抛出interruptedException异常。

1. 什么是线程安全问题？

线程安全是多线程编程时的计算机程序代码中的一个概念。在拥有共享数据的多条线程并行执行的程序中，线程安全的代码会通过同步机制保证各个线程都可以正常且正确的执行，不会出现数据污染等意外情况。

1. 什么是共享变量的内存可见性问题？

JMM(Java Memory Model),是一种基于计算机内存模型（定义了共享内存系统中多线程程序读写操作行为的规范），屏蔽了各种硬件和操作系统的访问差异的，保证了Java程序在各种平台下对内存的访问都能保证效果一致的机制及规范。保证共享内存的原子性、可见性、有序性。



上图描述了一个多线程执行场景。

线程 A 和线程 B 分别对主内存的变量进行读写操作。其中主内存中的变量为共享变量,也就是说此变量只此一份，多个线程间共享。但是线程不能直接读写主内存的共享变量，每个线程都有自己的工作内存，线程需要读写主内存的共享变量时需要先将该变量拷贝一份副本到自己的工作内存，然后在自己的工作内存中对该变量进行所有操作，线程工作内存对变量副本完成操作之后需要将结果同步至主内存。

那么问题来了，线程工作内存怎么知道什么时候又是怎样将数据同步到主内存呢？ 这里就轮到 JMM 出场了。 JMM 规定了何时以及如何做线程工作内存与主内存之间的数据同步。

1. 什么是Java中原子性操作？

对共享内存的操作必须是要么全部执行直到执行结束，且中间过程不能被任何外部因素打断，要么就不执行。

1. 什么是Java中的CAS操作,AtomicLong实现原理？

CAS的全称是Compare And Swap，即比较交换，当然还有一种说法：Compare And Set，调用原生CAS操作需要确定三个值：

CAS指令需要三个操作数，分别是内存地址（在Java内存模型中可以简单理解为主内存中变量的内存地址）、旧值（在Java内存模型中，可以理解工作内存中缓存的主内存的变量的值）和新值。CAS操作执行时，当且仅当主内存对应的值等于旧值时，处理器用新值去更新旧值，否则它就不执行更新。但是无论是否更新了主内存中的值，都会返回旧值，上述的处理过程是一个原子操作。

最常见的应该就是使用 AtomicXXX、以及在使用 Lock 相关的子类 的时候我们知道他们的底层运用了 CAS。AtomicLong中的源码：

public final boolean compareAndSet(long expect, long update) {  
 return *unsafe*.compareAndSwapLong(this, *valueOffset*, expect, update);  
}

调用了unsafe下的compareAndSwapInt方法，除了传递了我们传到此方法的两个参数之外，又传递了两个参数，这两个参数就是我上面说的实例和偏移地址，this代表是当前类的实例，即AtomicInteger类的实例，这个偏移地址又是什么鬼呢，说的简单点，就是确定我们需要修改的字段在实例的哪个位置。知道了实例，知道了我们的需要修改的字段是在实例的哪个位置，就可以确定这个字段了。不过，这个确定的过程不是在Java中做的，而是在更底层做的。

偏移地址是在本类的静态代码块中获得的

static {  
 try {  
 *valueOffset* = *unsafe*.objectFieldOffset  
 (AtomicLong.class.getDeclaredField("value"));  
 } catch (Exception ex) { throw new Error(ex); }  
}

unsafe.objectFieldOffset接收的是Field类型的参数，得到的就是对应字段的偏移地址了，这里就是获得value字段在本类，即AtomicLong中的偏移地址。

我们在来看看value字段的定义：

private volatile long value;

compareAndSwapInt和objectFieldOffset这两个方法由C++实现，最终会把对应的指令发送给CPU，这是可以保证原子性的

我们再来看看incrementAndGet的源码

public final long incrementAndGet() {  
 return *unsafe*.getAndAddLong(this, *valueOffset*, 1L) + 1L;  
}

getAndAddLong方法

public final long getAndAddLong(Object var1, long var2, long var4) {  
 long var6;  
 do {  
 var6 = this.getLongVolatile(var1, var2);  
 } while(!this.compareAndSwapLong(var1, var2, var6, var6 + var4));  
  
 return var6;  
}

incrementAndGet方法会调到用getAndAddInt方法，这里有三个参数：

var1：实例。

var2：偏移地址。

var4：需要自增的值，这里是1。

getAndAddInt方法内部有一个while循环，循环体内部根据实例和偏移地址获得对应的值，这里先称为A，再来看看while里面的判断内容，JDK和更底层进行通讯：嘿，我把实例和偏移地址给你，你帮我看下这个值是不是A，如果是的话，帮我修改成A+1，返回true，如果不是的话，返回false吧。

这里要思考一个问题：为什么需要while循环？

比如同时有两个线程执行到了getIntVolatile方法，拿到的值都是10，其中线程A执行native方法，修改成功，但是线程B就修改失败了啊，因为CAS操作是可以保证原子性的，所以线程B只能苦逼的再一次循环，这一次拿到的值是11，又去执行native方法，修改成功。

像这样的while循环，有一个高大上的称呼：CAS自旋。

让我们试想一下，如果现在并发真的很高很高，会出现什么事情？大量的线程在进行CAS自旋，这太浪费CPU了吧。所以在Java8之后，对原子操作类进行了一定的优化，这个我们后面再说。

CAS会出现的三大问题

虽然通过CAS操作可以很好的提高我们在处理数据的时候的效率，但是任然会出现许多问题。但是Java的开发团队已经为我们提供了一些处理方案，现在我们就来看看CAS有哪三大问题。

ABA问题

因为CAS需要在操作值的时候，检查值有没有发生变化，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，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。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1，那么A→B→A就会变成1A→2B→3A。从Java 1.5开始，JDK的Atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法的作用是首先检查当前引用是否等于预期引用，并且检查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。关于AtomicStampedReference的使用，有兴趣的小伙伴可以自行查看相关源码实现。

循环时间开销太大

在后期的文章我们会讲述自旋CAS，关于自旋CAS,因为后期关于锁的文章会具体描述，这里我就简单描述一下，在Java中有很多的并发框架都使用了自旋CAS来获取相应的锁，会一直循环直到获取到相应的锁后，然后执行相应的操作。那么当其自旋时CAS，会一直占用CPU的资源。如果自旋CAS长时间不成功，会给CPU带来非常大的执行开销。

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，自旋CAS就无法保证操作的原子性，这个时候就可以用锁。还有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如，有两个共享变量i＝2，j=a，合并一下ij=2a，然后用CAS来操作ij。从Java 1.5开始，JDK提供了AtomicReference类来保证引用对象之间的原子性，就可以把多个变量放在一个对象里来进行CAS操作。关于AtomicReference的使用，有兴趣的小伙伴可以自行查看相关源码实现。

1. 什么是Java指令重排序？

在计算机执行指令的顺序在经过程序编译器编译之后形成的指令序列，一般而言，这个指令序列是会输出确定的结果；以确保每一次的执行都有确定的结果。但是，一般情况下，CPU和编译器为了提升程序执行的效率，会按照一定的规则允许进行指令优化，在某些情况下，这种优化会带来一些执行的逻辑问题，主要的原因是代码逻辑之间是存在一定的先后顺序，在并发执行情况下，会发生二义性，即按照不同的执行逻辑，会得到不同的结果信息

为什么重排可以提高代码的执行效率？

大多数现代微处理器都会采用将指令乱序执行（out-of-order execution，简称OoOE或OOE）的方法，在条件允许的情况下，直接运行当前有能力立即执行的后续指令，避开获取下一条指令所需数据时造成的等待。通过乱序执行的技术，处理器可以大大提高执行效率。

除了处理器，常见的Java运行时环境的JIT编译器也会做指令重排序操作，即生成的机器指令与字节码指令顺序不一致。

数据依赖性

主要指不同的程序指令之间的顺序是不允许进行交换的，即可称这些程序指令之间存在数据依赖性。

哪些指令不允许重排？

主要的例子如下：

名称 代码示例 说明

写后读 a = 1;b = a; 写一个变量之后，再读这个位置。

写后写 a = 1;a = 2; 写一个变量之后，再写这个变量。

读后写 a = b;b = 1; 读一个变量之后，再写这个变量。

进过分析，发现这里每组指令中都有写操作，这个写操作的位置是不允许变化的，否则将带来不一样的执行结果。

编译器将不会对存在数据依赖性的程序指令进行重排，这里的依赖性仅仅指单线程情况下的数据依赖性；多线程并发情况下，此规则将失效

as-if-serial语义

不管怎么重排序（编译器和处理器为了提高并行度），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。

分析： 关键词是单线程情况下，必须遵守；其余的不遵守。

as-if-serial语义是啥？

as-if-serial语义的意思是，所有的动作（Action）都可以为了优化而被重排序，但是必须保证它们重排序后的结果和程序代码本身的应有结果是一致的。Java编译器、运行时和处理器都会保证单线程下的as-if-serial语义。

比如，为了保证这一语义，重排序不会发生在有数据依赖的操作之中。

double pi = 3.14; //A

double r = 1.0; //B

double area = pi \* r \* r; //C

分析代码：

A->C B->C； A，B之间不存在依赖关系； 故在单线程情况下， A与B的指令顺序是可以重排的，C不允许重排，必须在A和B之后。

as-if-serial语义把单线程程序保护了起来，遵守as-if-serial语义的编译器，runtime 和处理器共同为编写单线程程序的程序员创建了一个幻觉：单线程程序是按程序的顺序来执行的。as-if-serial语义使单线程程序员无需担心重排序会干扰他们，也无需担心内存可见性问题。核心点还是单线程，多线程情况下不遵守此原则。

Java的目标是成为一门平台无关性的语言，即Write once, run anywhere。但是不同硬件环境下指令重排序的规则不尽相同。例如，x86下运行正常的Java程序在IA64下就可能得到非预期的运行结果。为此，JSR-1337制定了Java内存模型(Java Memory Model, JMM)，旨在提供一个统一的可参考的规范，屏蔽平台差异性。从Java 5开始，Java内存模型成为Java语言规范的一部分。

根据Java内存模型中的规定，可以总结出以下几条happens-before规则。Happens-before的前后两个操作不会被重排序且后者对前者的内存可见。

程序次序法则：线程中的每个动作A都happens-before于该线程中的每一个动作B，其中，在程序中，所有的动作B都能出现在A之后。

监视器锁法则：对一个监视器锁的解锁 happens-before于每一个后续对同一监视器锁的加锁。

volatile变量法则：对volatile域的写入操作happens-before于每一个后续对同一个域的读写操作。

线程启动法则：在一个线程里，对Thread.start的调用会happens-before于每个启动线程的动作。

线程终结法则：线程中的任何动作都happens-before于其他线程检测到这个线程已经终结、或者从Thread.join调用中成功返回，或Thread.isAlive返回false。

中断法则：一个线程调用另一个线程的interrupt happens-before于被中断的线程发现中断。

终结法则：一个对象的构造函数的结束happens-before于这个对象finalizer的开始。

传递性：如果A happens-before于B，且B happens-before于C，则A happens-before于C。

Happens-before关系只是对Java内存模型的一种近似性的描述，它并不够严谨，但便于日常程序开发参考使用，关于更严谨的Java内存模型的定义和描述，请阅读JSR-133原文或Java语言规范章节17.4。

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final语义的扩展保证一个对象的构建方法结束前，所有final成员变量都必须完成初始化

Java内存模型关于重排序的规定，总结后如下表所示



表中"第二项操作"的含义是指，第一项操作之后的所有指定操作。如，普通读不能与其之后的所有volatile写重排序。另外，JMM也规定了上述volatile和同步块的规则尽适用于存在多线程访问的情景。例如，若编译器（这里的编译器也包括JIT，下同）证明了一个volatile变量只能被单线程访问，那么就可能会把它做为普通变量来处理。

留白的单元格代表允许在不违反Java基本语义的情况下重排序。例如，编译器不会对对同一内存地址的读和写操作重排序，但是允许对不同地址的读和写操作重排序。

首先我们基于一段代码的示例来分析，在多线程情况下，重排是否有不同结果信息：

class ReorderExample {

int a = 0;

boolean flag = false;

public void writer() {

a = 1; // 1

flag = true; // 2

}

public void reader() {

if (flag) { // 3

int i = a \* a; // 4

}

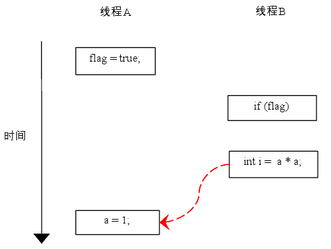
}

}

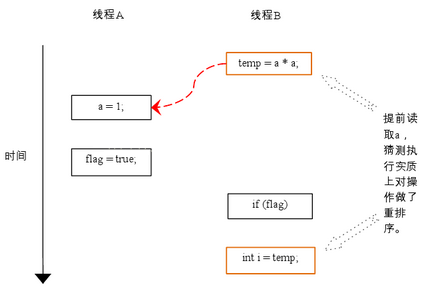
上述的代码，在单线程情况下，执行结果是确定的， flag=true将被reader的方法体中看到，并正确的设置结果。 但是在多线程情况下，是否还是只有一个确定的结果呢？

假设有A和B两个线程同时来执行这个代码片段， 两个可能的执行流程如下：

可能的流程1, 由于1和2语句之间没有数据依赖关系，故两者可以重排，在两个线程之间的可能顺序如下：



可能的流程2:， 在两个线程之间的语句执行顺序如下：



在程序中，操作3和操作4存在控制依赖关系。当代码中存在控制依赖性时，会影响指令序列执行的并行度。为此，编译器和处理器会采用猜测（Speculation）执行来克服控制相关性对并行度的影响。以处理器的猜测执行为例，执行线程B的处理器可以提前读取并计算a\*a，然后把计算结果临时保存到一个名为重排序缓冲（reorder buffer ROB）的硬件缓存中。当接下来操作3的条件判断为真时，就把该计算结果写入变量i中。从图中我们可以看出，猜测执行实质上对操作3和4做了重排序。重排序在这里破坏了多线程程序的语义。

与上面的例子类似的有：

在线程A中：

context = loadContext();

inited = true;

在线程B中：

while(!inited ){ //根据线程A中对inited变量的修改决定是否使用context变量

sleep(100);

} doSomethingwithconfig(context);

假设线程A中发生了指令重排序：

inited = true;

context = loadContext();

那么B中很可能就会拿到一个尚未初始化或尚未初始化完成的context，从而引发程序错误。

重排导致双重锁定的单例模式失效的例子

例子2：指令重排导致单例模式失效

我们都知道一个经典的懒加载方式的双重判断单例模式：

public class Singleton {

private static Singleton instance = null;

private Singleton() { }

public static Singleton getInstance() {

if(instance == null) {

synchronzied(Singleton.class) {

if(instance == null) {

instance = new Singleton(); //非原子操作

}

}

}

return instance;

}

}

似简单的一段赋值语句：instance= new Singleton()，但是它并不是一个原子操作，其实际上可以抽象为下面几条JVM指令：

memory =allocate(); //1：分配对象的内存空间

ctorInstance(memory); //2：初始化对象

instance =memory; //3：设置instance指向刚分配的内存地址

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

memory =allocate(); //1：分配对象的内存空间

instance =memory; //3：instance指向刚分配的内存地址，此时对象还未初始化

ctorInstance(memory); //2：初始化对象

可以看到指令重排之后，instance指向分配好的内存放在了前面，而这段内存的初始化被排在了后面。

在线程A执行这段赋值语句，在初始化分配对象之前就已经将其赋值给instance引用，恰好另一个线程进入方法判断instance引用不为null，然后就将其返回使用，导致出错。

解决方案：例子1中的inited和例子2中的instance以关键字volatile修饰之后，就会阻止JVM对其相关代码进行指令重排，这样就能够按照既定的顺序指执行。

1. Java中Volatile关键字的内存语义是什么？

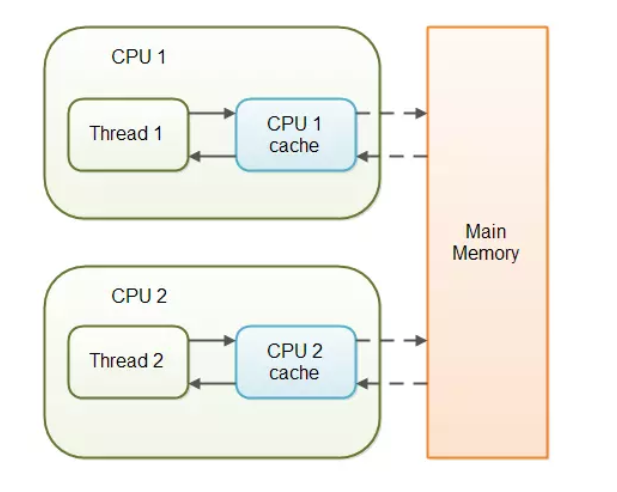
Java的关键字 volatile 用于将变量标记为“存储于主内存中”。更确切地说，对 volatile 变量的每次读操作都会直接从计算机的主存中读取，而不是从 cpu 缓存中读取；同样，每次对 volatile 变量的写操作都会直接写入到主存中，而不仅仅写入到 cpu 缓存里。

实际上，从 Java 5 开始关键字 volatile 除了能确保 volatile 变量直接从主存中进行读写，还有以下几个作用

可见性保证

关键字 volatile 能确保数据变化在线程之间的可见性。

在多线程的应用中多个线程对 non-volatile 变量进行操作，线程在对它们进行操作的时候为了提高性能会将变量从主存复制到 cpu 缓存中。如果你的电脑包含的 cpu 不止一个， 那么每个线程可能会运行于不同的 cpu 上。这意味着，不同线程会将变量复制到不同 cpu 的缓存里。如下图：

no-volatile 变量不能保证 Java 虚拟机（JVM）何时从主存中将数据读入cpu 缓存，也不能保证何时将数据从 cpu 缓存写入到主存中。这会带来一些问题，我将在下面解释。

no-volatile 变量不能保证 Java 虚拟机（JVM）何时从主存中将数据读入cpu 缓存，也不能保证何时将数据从 cpu 缓存写入到主存中。

想象一个场景，两个或两个以上线程可访问同一个共享对象，这个对象含有一个如下的计数器变量：

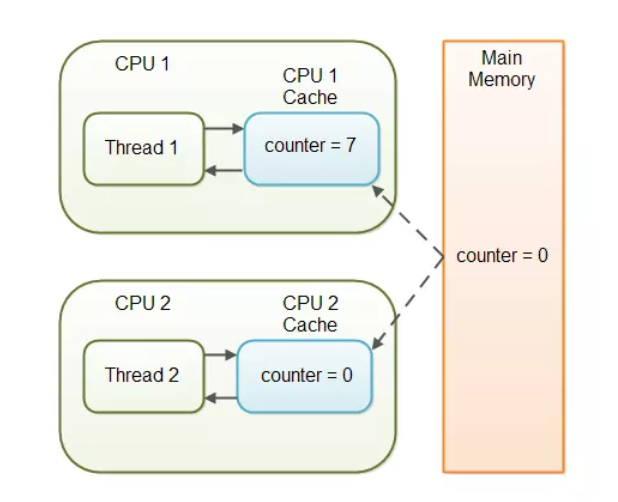
public class SharedObject{

public int counter = 0;

}

再假设有2个线程 Thread1 和 Thread2，只有 Thread1 能增大 counter ，而 Thread1 和 Thread2 都可以在任何时刻读取 counter 的值。

如果 counter 没被声明为 volatile ，将不能保证什么时候 counter 变量的值会从 cpu 缓存回写到主存内。也就是说，变量 counter 在 cpu 缓存中的值可能和主存内的不一致。 如下图：



这种由于线程还未将变量的值回写到主存而导致其他线程不能看到该变量的最新值的问题，称为可见性问题。一个线程的更新操作对其他线程不可见。

通过将变量 counter 声明为 volatile ，对其进行的所有写操作都会马上回写至主存中。同时，所有 counter 的读操作也将直接在主存中进行。声明方式如下：

public class SharedObject{

public volatile int counter = 0;

}

这样，将变量声明为 volatile 保证了写操作对其他线程的可见性。

当线程写入 volatile 时，不单单是将这个 volatile 写入主存中。这个线程在写此

volatile 变量之前改变的所有的变量也将刷新到主存中。当另一个线程读取这个 volatile 变量时，它也能从主存中读取到随 volatile 一起被刷入主存的其他所有变量。

看看这个例子：

Thread A:

sharedObject.nonVolatile = 123;

sharedObject.counter = sharedObject.counter + 1; // volatile

Thread B:

int counter = sharedObject.counter;

int nonVolatile = sharedObject.nonVolatile;

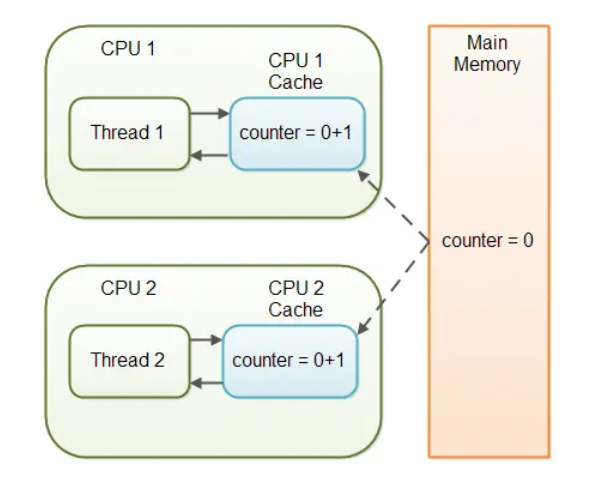
由于线程A 在写 volatile 的变量 sharedObject.counter 之前写 non-volatile 变量 sharedObject.nonVolatile，sharedObject.counter 和 sharedObject.nonVolatile 会在 写 sharedObject.counter 的时候一起写入到主存中。

由于线程B 开始时先读取 volatile 变量 sharedObject.counter， 那么 sharedObject.counter 和 sharedObject.nonVolatile 会直接从主存读取到供线程B 使用的 cpu 缓存中 。这个时候，线程B 读到的 sharedObject.nonVolatile 就是线程A 写入的新值。

volatile 并不能满足所有场景

即使关键字 volatile 能保证对它的所有读操作都是直接从主存中读取，但它不保证线程安全。

假设线程1 读取值为0的共享变量 counter 到它的 cpu 缓存中，增加值到1，但还没把改变的值回写到主存中。 线程2 接着也能从主存中读取值还是0这个 counter，并将其存入它自己的 cpu 缓存里。接着线程2 也将 counter 的值增加到1，同样也还没回写主存。这个场景如下图：



线程1 和线程2 现在就是切实的不同步。共享变量 counter 实际的值应该 2，但是每个线程在各自的 cpu 缓存中的值为 1，主存中的值还是 0。乱成一团了！即使最后线程都把它们持有的值回写到主存中，counter 的值也是错的。

例如你让一个volatile的integer自增（i++），其实要分成3步：1）读取volatile变量值到local； 2）增加变量的值；3）把local的值写回，让其它的线程可见。这3步的jvm指令为：

mov 0xc(%r10),%r8d ; Load

inc %r8d ; Increment

mov %r8d,0xc(%r10) ; Store

lock addl $0x0,(%rsp) ; StoreLoad Barrier

注意最后一步是内存屏障。

volatile关键字通过提供"内存屏障"的方式来防止指令被重排序，为了实现volatile的内存语义，编译器在生成字节码时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。

大多数的处理器都支持内存屏障的指令。

对于编译器来说，发现一个最优布置来最小化插入屏障的总数几乎不可能，为此，Java内存模型采取保守策略。下面是基于保守策略的JMM内存屏障插入策略：

在每个volatile写操作的前面插入一个StoreStore屏障。

在每个volatile写操作的后面插入一个StoreLoad屏障。

在每个volatile读操作的后面插入一个LoadLoad屏障。

在每个volatile读操作的后面插入一个LoadStore屏障。

内存屏障

内存屏障（Memory Barrier，或有时叫做内存栅栏，Memory Fence）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

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。

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。

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。

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。

有的处理器的重排序规则较严，无需内存屏障也能很好的工作，Java编译器会在这种情况下不放置内存屏障。

为了实现前面讨论的JSR-133的规定，Java编译器会这样使用内存屏障。

对 volatile 变量的读取和写入操作导致变量直接在主存中读写。从主存中读取和写入到主存中比在 cpu 缓存中代价更高 。访问 volatile 变量也阻止了常规的性能优化技术对指令的重排序。所以，你应该只在确实需要加强变量的可见性的时候使用 volatile。

1. Java中Synchronized关键字的内存语义是什么？

在多线程并发编程中synchronized一直是元老级角色，我们在开发过程中可以使用它来解决线程安全问题中提到的原子性，可见性，以及顺序性。很多人都会称呼它为重量级锁。但是，随着Java SE 1.6对synchronized进行了各种优化之后，有些情况下它就并不那么重了，Java SE 1.6中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，以及锁的存储结构和升级过程。

synchronized的三种应用方式：

　　synchronized有三种方式来加锁，分别是：方法锁，对象锁synchronized(this)，类锁synchronized(Demo.Class)。其中在方法锁层面可以有如下3种方式：

1. 修饰实例方法，作用于当前实例加锁，进入同步代码前要获得当前实例的锁

2. 静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁

3. 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。

synchronized括号后面的对象：

synchronized扩号后面的对象是一把锁，在java中任意一个对象都可以成为锁，简单来说，我们把object比喻是一个key，拥有这个key的线程才能执行这个方法，拿到这个key以后在执行方法过程中，这个key是随身携带的，并且只有一把。如果后续的线程想访问当前方法，因为没有key所以不能访问只能在门口等着，等之前的线程把key放回去。所以，synchronized锁定的对象必须是同一个，如果是不同对象，就意味着是不同的房间的钥匙，对于访问者来说是没有任何影响的。

Synchronized的实现原理

Java 虚拟机中的同步(Synchronization)基于进入和退出管程(Monitor)对象实现， 无论是显式同步(有明确的 monitorenter 和 monitorexit 指令,即同步代码块)还是隐式同步都是如此。在 Java 语言中，同步用的最多的地方可能是被 synchronized 修饰的同步方法。同步方法 并不是由 monitorenter 和 monitorexit 指令来实现同步的，而是由方法调用指令读取运行时常量池中方法的 ACC\_SYNCHRONIZED 标志来隐式实现的。

synchronized作用在类实例上字节码指令，可以看出调用了monitorenter和monitorexit

public class Synchronize{

public static void main(String[] args) {

synchronized (Synchronize.class){

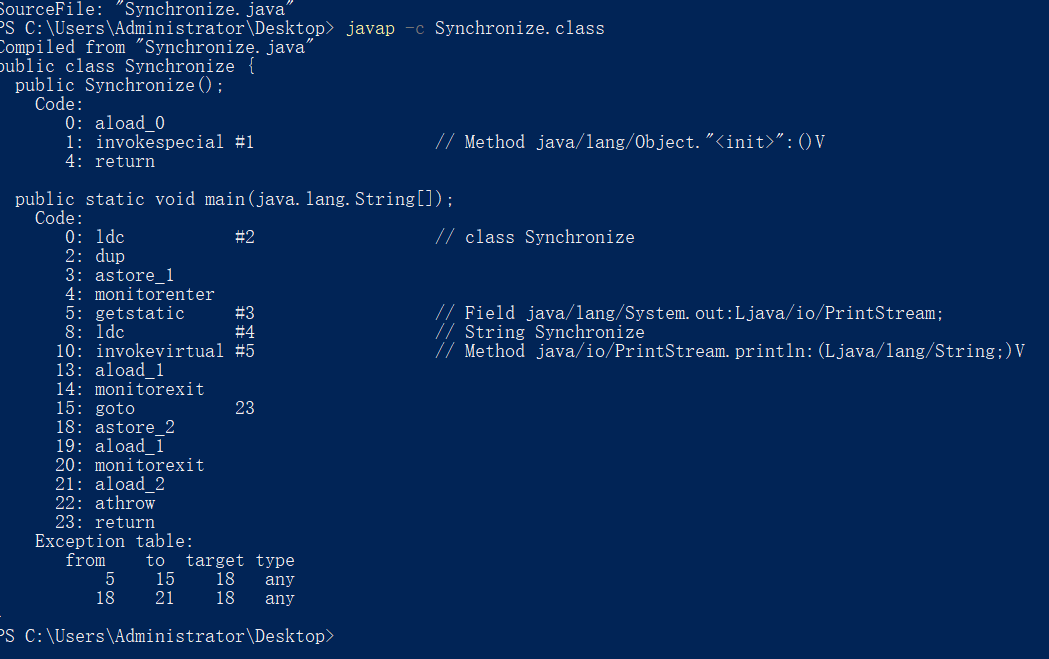
System.out.println("Synchronize");

}

}

}

使用javap -c Synchronize.class



修改以上代码，让synchronized作用在方法上，可以看到使用了ACC\_SYNCHRONIZED标志位

public class Synchronize{

public static void main(String[] args) {

print();

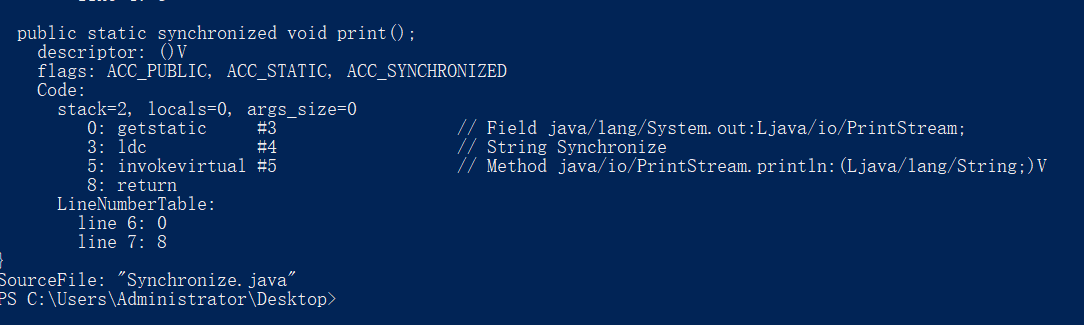
}

public static synchronized void print(){

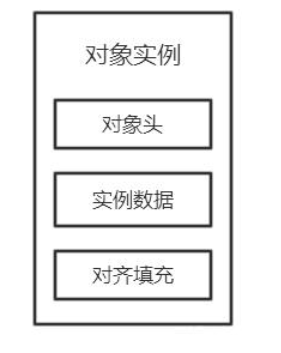
System.out.println("Synchronize");

}

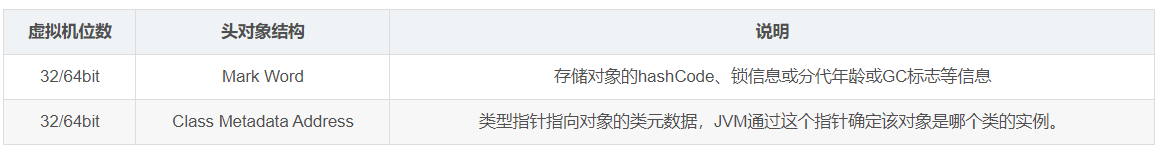
}



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



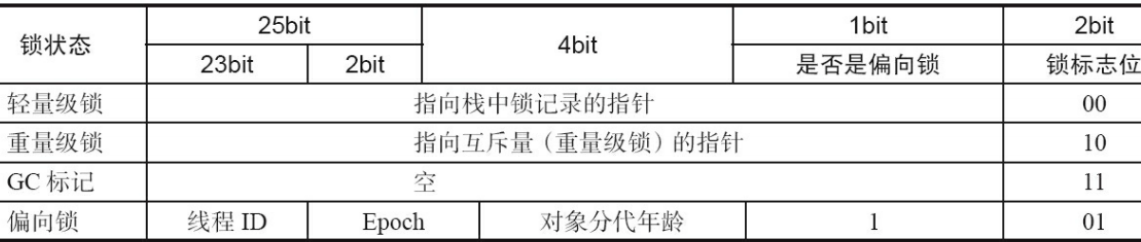
Java对象头，它实现synchronized的锁对象的基础，这点我们重点分析它，一般而言，synchronized使用的锁对象是存储在Java对象头里的，jvm中采用2个字来存储对象头(如果对象是数组则会分配3个字，多出来的1个字记录的是数组长度)，其主要结构是由Mark Word 和 Class Metadata Address 组成，其结构说明如下表：



其中Mark Word在默认情况下存储着对象的HashCode、分代年龄、锁标记位等以下是32位JVM的Mark Word默认存储结构



Mark Word里面存储的数据会随着锁标志位的变化而变化，Mark Word可能变化为存储以下4种情况



其中轻量级锁和偏向锁是Java 6 对 synchronized 锁进行优化后新增加的。这里我们主要分析一下重量级锁也就是通常说synchronized的对象锁，锁标识位为10，其中指针指向的是monitor对象（也称为管程或监视器锁）的起始地址。

Java中的每个对象都派生自Object类，而每个Java Object在JVM内部都有一个native的C++对象 oop/oopDesc进行对应。其次，线程在获取锁的时候，实际上就是获得一个监视器对象(monitor) ,monitor可以认为是一个同步对象，所有的Java对象是天生携带monitor，monitor可以与对象一起创建销毁或当线程试图获取对象锁时自动生成，但当一个 monitor 被某个线程持有后，它便处于锁定状态。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，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，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 WaitSe t集合中等待被唤醒。若当前线程执行完毕也将释放monitor(锁)并复位变量的值，以便其他线程进入获取monitor(锁)。

Java虚拟机对synchronized的优化

锁的状态总共有四种，无锁状态、偏向锁、轻量级锁和重量级锁。随着锁的竞争，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，关于重量级锁，前面我们已详细分析过，下面我们将介绍偏向锁和轻量级锁以及JVM的其他优化手段，这里并不打算深入到每个锁的实现和转换过程更多地是阐述Java虚拟机所提供的每个锁的核心优化思想，毕竟涉及到具体过程比较繁琐，如需了解详细过程可以查阅《深入理解Java虚拟机原理》。

偏向锁

偏向锁是Java 6之后加入的新锁，它是一种针对加锁操作的优化手段，经过研究发现，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，因此为了减少同一线程获取锁(会涉及到一些CAS操作,耗时)的代价而引入偏向锁。偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作，即获取锁的过程，这样就省去了大量有关锁申请的操作，从而也就提供程序的性能。所以，对于没有锁竞争的场合，偏向锁有很好的优化效果，毕竟极有可能连续多次是同一个线程申请相同的锁。但是对于锁竞争比较激烈的场合，偏向锁就失效了，因为这样场合极有可能每次申请锁的线程都是不相同的，因此这种场合下不应该使用偏向锁，否则会得不偿失，需要注意的是，偏向锁失败后，并不会立即膨胀为重量级锁，而是先升级为轻量级锁。下面我们接着了解轻量级锁。

轻量级锁

倘若偏向锁失败，虚拟机并不会立即升级为重量级锁，它还会尝试使用一种称为轻量级锁的优化手段(1.6之后加入的)，此时Mark Word 的结构也变为轻量级锁的结构。轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。

自旋锁

轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。这是基于在大多数情况下，线程持有锁的时间都不会太长，如果直接挂起操作系统层面的线程可能会得不偿失，毕竟操作系统实现线程之间的切换时需要从用户态转换到核心态，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，因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。

锁消除

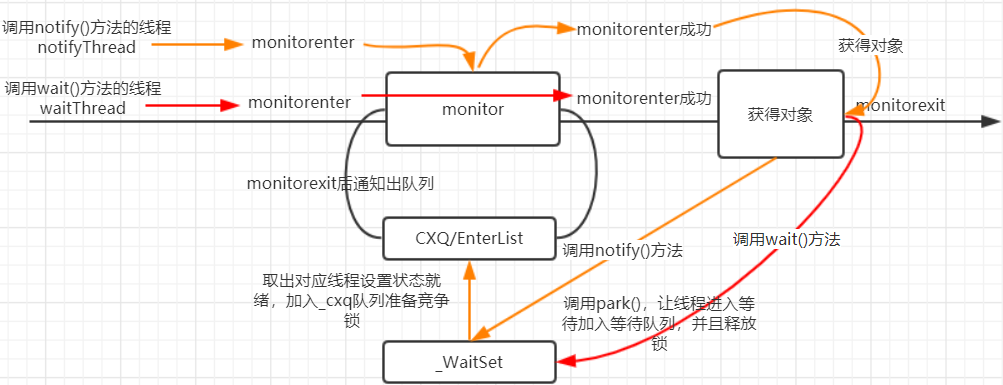
消除锁是虚拟机另外一种锁的优化，这种优化更彻底，Java虚拟机在JIT编译时(可以简单理解为当某段代码即将第一次被执行时进行编译，又称即时编译)，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过这种方式消除没有必要的锁，可以节省毫无意义的请求锁时间，如下StringBuffer的append是一个同步方法，但是在add方法中的StringBuffer属于一个局部变量，并且不会被其他线程所使用，因此StringBuffer不可能存在共享资源竞争的情景，JVM会自动将其锁消除。

重量级锁：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。主要是，当系统检查到锁是重量级锁之后，会把等待想要获得锁的线程进行阻塞，被阻塞的线程不会消耗cup。但是阻塞或者唤醒一个线程时，都需要操作系统来帮忙，这就需要从用户态转换到内核态，而转换状态是需要消耗很多时间的，有可能比用户执行代码的时间还要长。这就是说为什么重量级线程开销很大的。

有关锁升级的具体内容可以参考<https://segmentfault.com/a/1190000017255044>

wait和notify的原理：

　　调用wait方法，首先会获取监视器锁，获得成功以后，会让当前线程进入等待状态进入等待队列并且释放锁；然后当其他线程调用notify或者notifyall以后，会通知等待线程可以醒了，而执行完notify方法以后，并不会立马唤醒线程，原因是当前的线程仍然持有这把锁，处于等待状态的线程无法获得锁。必须要等到当前的线程执行完按monitorexit指令以后，也就是锁被释放以后，处于等待队列中的线程就可以开始竞争锁了。



wait和notify为什么需要在synchronized里面:

wait方法的语义有两个，一个是释放当前的对象锁、另一个是使得当前线程进入阻塞队列， 而这些操作都和监视器是相关的，所以wait必须要获得一个监视器锁。

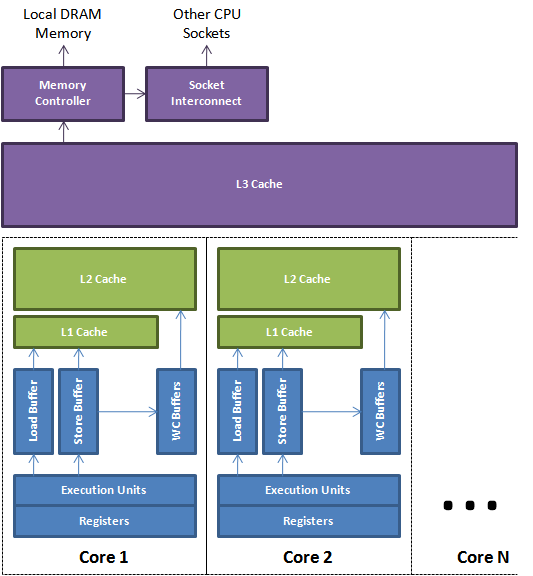
而对于notify来说也是一样，它是唤醒一个线程，既然要去唤醒，首先得知道它在哪里？所以就必须要找到这个对象获取到这个对象的锁，然后到这个对象的等待队列中去唤醒一个线程。

1. 什么是伪共享,为何会出现，以及如何避免？

缓存的存储方式，是以缓存行(Cache Line)为单位的。一般缓存行的大小是64字节。这意味着，小于64字节的变量，是有可能存在于同一条缓存行的。例如变量X大小32字节，变量Y大小32字节，那么他们有可能会存在于一条缓存行上。

伪共享，就是多个线程同时修改共享在同一个缓存行里的独立变量，无意中影响了性能。

伪共享示意图



上图说明了伪共享的问题。

在核心1上运行的线程想更新变量X，同时核心2上的线程想要更新变量Y，不幸的是，这两个变量在同一个缓存行中。

每个线程都要去竞争缓存行的所有权来更新变量。

如果核心1获得了所有权，缓存子系统将会使核心2中对应的缓存行失效。

当核心2获得了所有权然后执行更新操作，核心1就要使自己对应的缓存行失效。

这会来来回回的经过L3缓存，大大影响了性能。如果互相竞争的核心位于不同的插槽，就要额外横跨插槽连接，问题可能更加严重。

JDK1.8之前解决方式-padding方式

在JDK1.8以前，我们一般是在属性间增加长整型变量来分隔每一组属性。

被操作的每一组属性占的字节数加上前后填充属性所占的字节数，不小于一个cache line的字节数就可以达到要求：（通过填充变量，使不相关的变量分开）

public class DataPadding{

long a1,a2,a3,a4,a5,a6,a7,a8;//防止与前一个对象产生伪共享

int value;

long modifyTime;

long b1,b2,b3,b4,b5,b6,b7,b8;//防止不相关变量伪共享;

boolean flag;

long c1,c2,c3,c4,c5,c6,c7,c8;//

long createTime;

char key;

long d1,d2,d3,d4,d5,d6,d7,d8;//防止与下一个对象产生伪共享

}

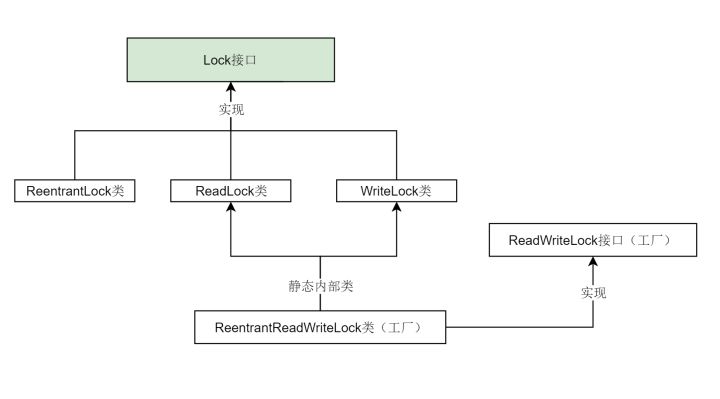
JDK1.8之后解决方式- Contended注解方式

在JDK1.8中，新增了一种注解@sun.misc.Contended，来使各个变量在Cache line中分隔开。

注意，jvm需要添加参数-XX:-RestrictContended才能开启此功能

用时，可以在类前或属性前加上此注释。

1. 什么是可重入锁、乐观锁、悲观锁、公平锁、非公平锁、独占锁、共享锁？



可重入锁的字面意思是“可以重新进入的锁”，即允许同一个线程多次获取同一把锁。比如一个递归函数里有加锁操作，递归过程中这个锁会阻塞自己吗？如果不会，那么这个锁就是可重入锁（因为这个原因可重入锁也叫做递归锁）。Java里只要以Reentrant开头命名的锁都是可重入锁，而且JDK提供的所有现成的Lock实现类，包括synchronized关键字锁都是可重入的。

悲观锁（Pessimistic Lock）, 就是很悲观，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。这样别人想拿数据就被挡住，直到悲观锁被释放。

乐观锁（Optimistic Lock）, 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，不会上锁！但是如果想要更新数据，则会在更新前检查在读取至更新这段时间别人有没有修改过这个数据。如果修改过，则重新读取，再次尝试更新，循环上述步骤直到更新成功（当然也允许更新失败的线程放弃操作）。

公平锁和非公平锁：如果多个线程申请一把公平锁，那么当锁释放的时候，先申请的先得到，非常公平。显然如果是非公平锁，后申请的线程可能先获取到锁，是随机或者按照其他优先级排序的。对ReentrantLock类而言，通过构造函数传参可以指定该锁是否是公平锁，默认是非公平锁。一般情况下，非公平锁的吞吐量比公平锁大，如果没有特殊要求，优先使用非公平锁。

独享锁：该锁每一次只能被一个线程所持有。

共享锁：该锁可被多个线程共有，典型的就是ReentrantReadWriteLock里的读锁，它的读锁是可以被共享的，但是它的写锁确每次只能被独占。

另外读锁的共享可保证并发读是非常高效的，但是读写和写写，写读都是互斥的。

独享锁与共享锁也是通过AQS来实现的，通过实现不同的方法，来实现独享或者共享。

对于Synchronized而言，当然是独享锁。

互斥锁：在访问共享资源之前对进行加锁操作，在访问完成之后进行解锁操作。 加锁后，任何其他试图再次加锁的线程会被阻塞，直到当前进程解锁。

如果解锁时有一个以上的线程阻塞，那么所有该锁上的线程都被编程就绪状态， 第一个变为就绪状态的线程又执行加锁操作，那么其他的线程又会进入等待。 在这种方式下，只有一个线程能够访问被互斥锁保护的资源

读写锁：读写锁既是互斥锁，又是共享锁，read模式是共享，write是互斥(排它锁)的。

读写锁有三种状态：读加锁状态、写加锁状态和不加锁状态

读写锁在Java中的具体实现就是ReadWriteLock

一次只有一个线程可以占有写模式的读写锁，但是多个线程可以同时占有读模式的读写锁。

只有一个线程可以占有写状态的锁，但可以有多个线程同时占有读状态锁，这也是它可以实现高并发的原因。当其处于写状态锁下，任何想要尝试获得锁的线程都会被阻塞，直到写状态锁被释放；如果是处于读状态锁下，允许其它线程获得它的读状态锁，但是不允许获得它的写状态锁，直到所有线程的读状态锁被释放；为了避免想要尝试写操作的线程一直得不到写状态锁，当读写锁感知到有线程想要获得写状态锁时，便会阻塞其后所有想要获得读状态锁的线程。所以读写锁非常适合资源的读操作远多于写操作的情况。

1. 讲讲ThreadLocal 的实现原理？ThreadLocal 作为变量的线程隔离方式，其内部是如何做的？

ThreadLocal是JDK包提供的，它提供线程本地变量，如果创建一乐ThreadLocal变量，那么访问这个变量的每个线程都会有这个变量的一个副本，在实际多线程操作的时候，操作的是自己本地内存中的变量，从而规避了线程安全问题。

ThreadLocal 使用例子：

package test;

public class ThreadLocalTest {

static ThreadLocal<String> localVar = new ThreadLocal<>();

static void print(String str) {

//打印当前线程中本地内存中本地变量的值

System.out.println(str + " :" + localVar.get());

//清除本地内存中的本地变量

localVar.remove();

}

public static void main(String[] args) {

Thread t1 = new Thread(new Runnable() {

@Override

public void run() {

//设置线程1中本地变量的值

localVar.set("localVar1");

//调用打印方法

print("thread1");

//打印本地变量

System.out.println("after remove : " + localVar.get());

}

});

Thread t2 = new Thread(new Runnable() {

@Override

public void run() {

//设置线程1中本地变量的值

localVar.set("localVar2");

//调用打印方法

print("thread2");

//打印本地变量

System.out.println("after remove : " + localVar.get());

}

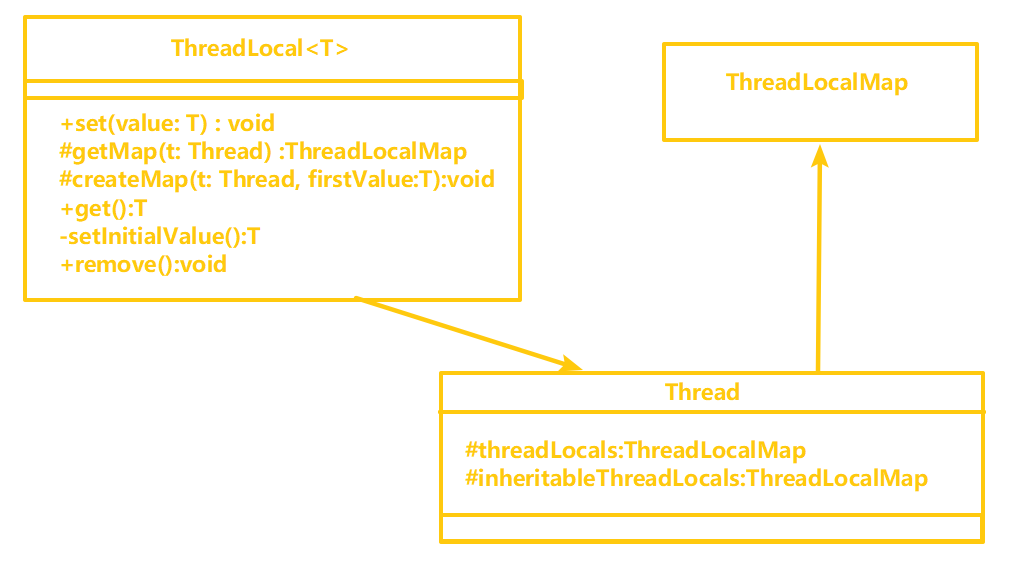
});

t1.start();

t2.start();

}

}



从图中可知：Thread类中有两个变量threadLocals和inheritableThreadLocal。ThreadLocalMap可以发现实际上它类似于一个HashMap。在默认情况下，每个线程中的这两个变量都为null，

只有当线程第一次调用ThreadLocal的set或者get方法的时候才会创建他们。ThreadLocal类型的本地变量是存放在具体的线程空间上，其本身相当于一个装载本地变量的工具壳，通过set方法将value添加到调用线程的threadLocals中，当调用线程调用get方法时候能够从它的threadLocals中取出变量。如果调用线程一直不终止，那么这个本地变量将会一直存放在他的threadLocals中，所以不使用本地变量的时候需要调用remove方法将threadLocals中删除不用的本地变量。

### **set 方法**

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

在上述源码中，首先会通过 getMap() 方法来获取当前线程中的 ThreadLocal 数据。获取的方法就是：直接去当前Thread t 中访问。因为在 Thread 类中有一个成员变量 ThreadLocal.ThreadLocalMap threadLocals = null;专门用于存储线程的 ThreadLocal 数据。这时候如果 threadLocals 为 null 的时候，就调用 createMap(t, value); 进行初始化，并把数据放进去，这个构造方法就在上面

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

接下来就分析一下上面中的 map.set(this, value) 的 set 方法，需要说明一下的是这个方法是在内部类ThreadLocalMap里面。

### **get 方法**

在get方法的实现中，首先获取当前调用者线程，如果当前线程的threadLocals不为null，就直接返回当前线程绑定的本地变量值，否则执行setInitialValue方法初始化threadLocals变量。在setInitialValue方法中，类似于set方法的实现，都是判断当前线程的threadLocals变量是否为null，是则添加本地变量（这个时候由于是初始化，所以添加的值为null），否则创建threadLocals变量，同样添加的值为null。

public T get() {  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 if (map != null) {  
 ThreadLocalMap.Entry e = map.getEntry(this);  
 if (e != null) {  
 @SuppressWarnings("unchecked")  
 T result = (T)e.value;  
 return result;  
 }  
 }  
 return setInitialValue();  
}  
  
*/\*\*  
 \* Variant of set() to establish initialValue. Used instead  
 \* of set() in case user has overridden the set() method.  
 \*  
 \** ***@return*** *the initial value  
 \*/*private T setInitialValue() {  
 T value = initialValue();  
 Thread t = Thread.*currentThread*();  
 ThreadLocalMap map = getMap(t);  
 if (map != null)  
 map.set(this, value);  
 else  
 createMap(t, value);  
 return value;  
}

remove

remove方法判断该当前线程对应的threadLocals变量是否为null，不为null就直接删除当前线程中指定的threadLocals变量

public void remove() {  
 ThreadLocalMap m = getMap(Thread.*currentThread*());  
 if (m != null)  
 m.remove(this);  
}

安全泄漏问题

static class ThreadLocalMap {  
  
 */\*\*  
 \* The entries in this hash map extend WeakReference, using  
 \* its main ref field as the key (which is always a  
 \* ThreadLocal object). Note that null keys (i.e. entry.get()  
 \* == null) mean that the key is no longer referenced, so the  
 \* entry can be expunged from table. Such entries are referred to  
 \* as "stale entries" in the code that follows.  
 \*/* static class Entry extends WeakReference<ThreadLocal<?>> {  
 */\*\* The value associated with this ThreadLocal. \*/* Object value;  
  
 Entry(ThreadLocal<?> k, Object v) {  
 super(k);  
 value = v;  
 }  
 }

上面我们知道ThreadLocalMap内部实际上是一个Entry数组，当前ThreadLocal的引用k被传递给WeakReference的构造函数，所以ThreadLocalMap中的key为ThreadLocal的弱引用。当一个线程调用ThreadLocal的set方法设置变量的时候，当前线程的ThreadLocalMap就会存放一个记录，这个记录的key值为ThreadLocal的弱引用，value就是通过set设置的值。如果当前线程一直存在且没有调用该ThreadLocal的remove方法，如果这个时候别的地方还有对ThreadLocal的引用，那么当前线程中的ThreadLocalMap中会存在对ThreadLocal变量的引用和value对象的引用，是不会释放的，就会造成内存泄漏。考虑这个ThreadLocal变量没有其他强依赖，如果当前线程还存在，由于线程的ThreadLocalMap里面的key是弱引用，所以当前线程的ThreadLocalMap里面的ThreadLocal变量的弱引用在gc的时候就被回收，但是对应的value还是存在的这就可能造成内存泄漏(因为这个时候ThreadLocalMap会存在key为null但是value不为null的entry项)。

总结：THreadLocalMap中的Entry的key使用的是ThreadLocal对象的弱引用，在没有其他地方对ThreadLoca依赖，ThreadLocalMap中的ThreadLocal对象就会被回收掉，但是对应的value不会被回收，这个时候Map中就可能存在key为null但是value不为null的项，这需要实际的时候使用完毕及时调用remove方法避免内存泄漏。

1. 说说InheritableThreadLocal 的实现原理？

public class InheritableThreadLocal<T> extends ThreadLocal<T> {

protected T childValue(T parentValue) {

return parentValue;

}

ThreadLocalMap getMap(Thread t) {

return t.inheritableThreadLocals;

}

void createMap(Thread t, T firstValue) {

t.inheritableThreadLocals = new ThreadLocalMap(this, firstValue);

}

}

从上面代码可以看出，InheritableThreadLocal类继承了ThreadLocal类，并重写了childValue、getMap、createMap三个方法。其中createMap方法在被调用（当前线程调用set方法时得到的map为null的时候需要调用该方法）的时候，创建的是inheritableThreadLocal而不是threadLocals。同理，getMap方法在当前调用者线程调用get方法的时候返回的也不是threadLocals而是inheritableThreadLocal。

1. InheritableThreadLocal 是如何弥补 ThreadLocal 不支持继承的特性？

说到InheritableThreadLocal线程间传值实现原理还要从Thread类说起

public class Thread implements Runnable {

......(其他源码)

/\*

\* 当前线程的ThreadLocalMap，主要存储该线程自身的ThreadLocal

\*/

ThreadLocal.ThreadLocalMap threadLocals = null;

/\*

\* InheritableThreadLocal，自父线程集成而来的ThreadLocalMap，

\* 主要用于父子线程间ThreadLocal变量的传递

\* 本文主要讨论的就是这个ThreadLocalMap

\*/

ThreadLocal.ThreadLocalMap inheritableThreadLocals = null;

......(其他源码)}

Thread类中包含 threadLocals 和 inheritableThreadLocals 两个变量，其中 inheritableThreadLocals 即主要存储可自动向子线程中传递的ThreadLocal.ThreadLocalMap。

接下来看一下父线程创建子线程的流程，我们从最简单的方式说起：

public Thread() {

init(null, null, "Thread-" + nextThreadNum(), 0);

}

/\*\*

\* 默认情况下，设置inheritThreadLocals可传递

\*/

private void init(ThreadGroup g, Runnable target, String name,

long stackSize) {

init(g, target, name, stackSize, null, true);

}

private void init(ThreadGroup g, Runnable target, String name,

long stackSize, AccessControlContext acc,

boolean inheritThreadLocals) {

......（其他代码）

if (inheritThreadLocals && parent.inheritableThreadLocals != null)

this.inheritableThreadLocals =

ThreadLocal.createInheritedMap(parent.inheritableThreadLocals);

......（其他代码）

}

可以看到，采用默认方式产生子线程时，inheritThreadLocals=true；若此时父线程inheritableThreadLocals不为空，则将父线程inheritableThreadLocals传递至子线程。

让我们继续追踪createInheritedMap：

private ThreadLocalMap(ThreadLocalMap parentMap) {  
 Entry[] parentTable = parentMap.table;  
 int len = parentTable.length;  
 setThreshold(len);  
 table = new Entry[len];  
  
 for (int j = 0; j < len; j++) {  
 Entry e = parentTable[j];  
 if (e != null) {  
 @SuppressWarnings("unchecked")  
 ThreadLocal<Object> key = (ThreadLocal<Object>) e.get();  
 if (key != null) {  
 Object value = key.childValue(e.value);  
 Entry c = new Entry(key, value);  
 int h = key.threadLocalHashCode & (len - 1);  
 while (table[h] != null)  
 h = *nextIndex*(h, len);  
 table[h] = c;  
 size++;  
 }  
 }  
 }  
}

1. CyclicBarrier内部的实现与 CountDownLatch 有何不同？

CountDownLatch是一个倒计时计数器，它可以让某一个线程等待多个线程执行结束后才开始执行。CountdownLatch end = new CountDownLatch(10);计数量为10，每次执行end.countDown()，就会减1。在主线程执行end.wait()后等待，当多个线程执行end.countDown()倒计时结束后主线程才执行下去。

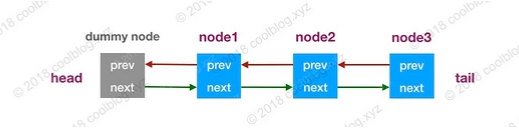
Public CyclicBarrier(int parties,Runnable barrierAction)

当计数完成后执行barrierAction

CyclicBarrier cyclic = new CyclicBarrier(N,new BarrierRun());

cyclic.wait()当有parties个线程执行cyclic.wait()后会触发BarrierRun线程执行。

CountDownLatch 的同步功能是基于 AQS 实现的，CountDownLatch 使用 AQS 中的 state 成员变量作为计数器。在 state 不为0的情况下，凡是调用 await 方法的线程将会被阻塞，并被放入 AQS 所维护的同步队列中进行等待。大致示意图如下：



每个阻塞的线程都会被封装成节点对象，节点之间通过 prev 和 next 指针形成同步队列。初始情况下，队列的头结点是一个虚拟节点。该节点仅是一个占位符，没什么特别的意义。每当有一个线程调用 countDown 方法，就将计数器 state--。当 state 被减至0时，队列中的节点就会按照 FIFO 顺序被唤醒，被阻塞的线程即可恢复运行。

CountDownLatch 本身的原理并不难理解，不过如果大家想深入理解 CountDownLatch 的实现细节，那么需要先去学习一下 AQS 的相关原理。CountDownLatch 是基于 AQS 实现的，所以理解 AQS 是学习 CountDownLatch 的前置条件。

CyclicBarrier 并没有直接通过 AQS 实现同步功能，而是在重入锁 ReentrantLock 的基础上实现的。在 CyclicBarrier 中，线程访问 await 方法需先获取锁才能访问。在最后一个线程访问 await 方法前，其他线程进入 await 方法中后，会调用 Condition 的 await 方法进入等待状态。在最后一个线程进入 CyclicBarrier await 方法后，该线程将会调用 Condition 的 signalAll 方法唤醒所有处于等待状态中的线程。同时，最后一个进入 await 的线程还会重置 CyclicBarrier 的状态，使其可以重复使用。

在创建 CyclicBarrier 对象时，需要转入一个值，用于初始化 CyclicBarrier 的成员变量 parties，该成员变量表示屏障拦截的线程数。当到达屏障的线程数小于 parties 时，这些线程都会被阻塞住。当最后一个线程到达屏障后，此前被阻塞的线程才会被唤醒。

其源码分析参考<https://segmentfault.com/a/1190000014818613>

1. 随机数生成器 Random 类如何使用 CAS 算法保证多线程下新种子的唯一性？

protected int next(int bits) {  
 long oldseed, nextseed;  
 AtomicLong seed = this.seed;  
 do {  
 oldseed = seed.get();  
 nextseed = (oldseed \* *multiplier* + *addend*) & *mask*;  
 } while (!seed.compareAndSet(oldseed, nextseed));  
 return (int)(nextseed >>> (48 - bits));  
}

在单线程情况下每次调用nextInt都是根据老的种子计算出来新的种子，这是可以保证随机数产生的随机性的。但是在多线程下多个线程可能都拿同一个老的 种子去执行步骤计算新的种子，这会导致多个线程产生的新种子是一样的，由于算法是固定的，所以会导致多个线程产生相同的随机值，这并不是 我们想要的。所以要保证原子性，也就是说多个线程在根据同一个老种子计算新种子时候，第一个线程的新种子计算出来后，第二个线程要丢弃自己老的 种子，要使用第一个线程的新种子来计算自己的新种子，依次类推，只有保证了这个，才能保证多线程下产生的随机数是随机的。

Random函数使用一个原子变量AtomicLong 达到了这个效果。总结下：每个Random实例里面有一个原子性的种子变量用来记录当前的种子的值，当要生成新的随机数时候要根据当前种子计算新的种子并更新回原子变量。 多线程下使用单个Random实例生成随机数时候，多个线程同时计算随机数计算新的种子时候多个线程会竞争同一个原子变量的更新操作，由于原子变量的更新 是CAS操作，同时只有一个线程会成功，所以会造成大量线程进行自旋重试，这是会降低并发性能的，所以ThreadLocalRandom应运而生。

1. ThreadLocalRandom 是如何利用 ThreadLocal 的原理来解决 Random 的局限性？

分析下ThreadLocalRandom的实现原理。类似ThreadLocal类，ThreadLocal的出现就是为了解决多线程访问一个变量时候需要进行同步的问题，让每一个线程拷贝一份变量， 每个线程对变量进行操作时候实际是操作自己本地内存里面的拷贝，从而避免了对共享变量进行同步。实际上ThreadLocalRandom的实现也是这个 原理，Random的缺点是多个线程会使用原子性种子变量，会导致对原子变量更新的竞争(若看到段有疑问的请不要停下来自己想,看下段)

那么如果每个线程维护自己的一个种子变量，每个线程生成随机数时候根据自己老的种子计算新的种子，并使用新种子更新老的种子，然后根据新种子计算随机数，就不会存在竞争问题，这会大大提高并发性能

1. Spring 框架中如何使用 ThreadLocal 实现 request scope 作用域 Bean？
2. 并发包中锁的实现底层（对AQS的理解）？

<https://segmentfault.com/a/1190000014721183>

<https://www.cnblogs.com/micrari/p/6937995.html>

ReentrantLock：ReentrantLock类使用AQS同步状态来保存锁重复持有的次数。当锁被一个线程获取时，ReentrantLock也会记录下当前获得锁的线程标识，以便检查是否是重复获取，以及当错误的线程试图进行解锁操作时检测是否存在非法状态异常。ReentrantLock也使用了AQS提供的ConditionObject，还向外暴露了其它监控和监测相关的方法。

ReentrantReadWriteLock：ReentrantReadWriteLock类使用AQS同步状态中的16位来保存写锁持有的次数，剩下的16位用来保存读锁的持有次数。WriteLock的构建方式同ReentrantLock。ReadLock则通过使用acquireShared方法来支持同时允许多个读线程。

　　Semaphore：Semaphore类（信号量）使用AQS同步状态来保存信号量的当前计数。它里面定义的acquireShared方法会减少计数，或当计数为非正值时阻塞线程；tryRelease方法会增加计数，在计数为正值时还要解除线程的阻塞。

　　CountDownLatch：CountDownLatch类使用AQS同步状态来表示计数。当该计数为0时，所有的acquire操作（对应到CountDownLatch中就是await方法）才能通过。

　　FutureTask：FutureTask类使用AQS同步状态来表示某个异步计算任务的运行状态（初始化、运行中、被取消和完成）。设置（FutureTask的set方法）或取消（FutureTask的cancel方法）一个FutureTask时会调用AQS的release操作，等待计算结果的线程的阻塞解除是通过AQS的acquire操作实现的。

　　SynchronousQueues：SynchronousQueues类使用了内部的等待节点，这些节点可以用于协调生产者和消费者。同时，它使用AQS同步状态来控制当某个消费者消费当前一项时，允许一个生产者继续生产，反之亦然。

1. 讲讲独占锁 ReentrantLock 原理？

Lock()获得锁，如果锁已被占用，则等待

* lockInterruptibly()获得锁，优先响应中断
* tryLock()尝试获得锁，如果成功返回true，如果失败则返回false
* tryLock(long time,TimeUnit unit)在给定的时间尝试获得锁
* unlock释放锁

Condition condition = Lock.newCondition()

Await()会使当前线程等待，并释放当前锁，当其他线程使用signal()或者signalAll(),线程会重新尝试获得锁并在获得锁后继续执行下去，或者被线程中断时也能跳出等地啊。

awaitUninterruptibly()与await()方法一致，不过不会再等待时响应中断

Singal()唤醒一个等待中的线程

1. 谈谈读写锁 ReentrantReadWriteLock 原理？

读不阻塞

写阻塞

ReentrantReadWriteLock readWriteLock = new ReentrantReadWriteLock();

Lock readLock = readWriteLock.readLock();

Lock writeLock = readWriteLock.writeLock();

<https://www.cnblogs.com/xiaoxi/p/9140541.html>

1. StampedLock 锁原理的理解？

StampedLock是Java 8新增的一个读写锁，它是对ReentrantReadWriteLock的改进。StampedLock的同步状态包含了一个版本和模式，获取锁的方法返回一个stamp表示这个锁的状态；而这些方法的 "try" 版本返回一个特殊值0表示获取锁失败。锁释放和转换的方法需要stamp作为参数，如果stamp不符合锁的同步状态就会失败。StampedLock提供了三种模式的控制：

独占写模式。writeLock方法可能会在获取共享状态时阻塞，如果成功获取锁，返回一个stamp，它可以作为参数被用在unlockWrite方法中以释放写锁。tryWriteLock的超时与非超时版本都被提供使用。当写锁被获取，那么没有读锁能够被获取并且所有的乐观读锁验证都会失败。

悲观读模式。readLock方法可能会在获取共享状态时阻塞，如果成功获取锁，返回一个stamp，它可以作为参数被用在unlockRead方法中以释放读锁。tryReadLock的超时与非超时版本都被提供使用。

乐观读模式。tryOptimisticRead方法只有当写锁没有被获取时会返回一个非0的stamp。在获取这个stamp后直到调用validate方法这段时间，如果写锁没有被获取，那么validate方法将会返回true。这个模式可以被认为是读锁的一个弱化版本，因为它的状态可能随时被写锁破坏。这个乐观模式的主要是为一些很短的只读代码块的使用设计，它可以降低竞争并且提高吞吐量。但是，它的使用本质上是很脆弱的。乐观读的代码区域应当只读取共享数据并将它们储存在局部变量中以待后来使用，当然在使用前要先验证这些数据是否过期，这可以使用前面提到的validate方法。在乐观读模式下的数据读取可能是非常不一致的过程，因此只有当你对数据的表示很熟悉并且重复调用validate方法来检查数据的一致性时使用此模式。例如，当先读取一个对象或者数组引用，然后访问它的字段、元素或者方法之一时上面的步骤都是需要的。

这个类还提供了在三种模式之间转换的辅助方法。例如，tryConvertToWriteLock方法尝试"提升"一个模式，如果已经获取了读锁并且此时没有其他线程获取读锁，那么这个方法返回一个合法的写stamp。这些方法被设计来帮助减少以“重试为主”设计时发生的代码代码膨胀。

下面的类中描述了一些StampedLock的常用用法，它主要操作一个简单的二维点。这个示例在没有异常会抛出的情况下依然沿用使用try-catch块的惯例。

class Point {

// 成员变量

private double x, y;

// 锁实例

private final StampedLock sl = new StampedLock();

// 排它锁-写锁（writeLock）

void move(double deltaX, double deltaY) {

long stamp = sl.writeLock();

try {

x += deltaX;

y += deltaY;

} finally {

sl.unlockWrite(stamp);

}

}

// 一个只读方法

// 其中存在乐观读锁到悲观读锁的转换

double distanceFromOrigin() {

// 尝试获取乐观读锁

long stamp = sl.tryOptimisticRead();

// 将全部变量拷贝到方法体栈内

double currentX = x, currentY = y;

// 检查在获取到读锁stamp后，锁有没被其他写线程抢占

if (!sl.validate(stamp)) {

// 如果被抢占则获取一个共享读锁（悲观获取）

stamp = sl.readLock();

try {

// 将全部变量拷贝到方法体栈内

currentX = x;

currentY = y;

} finally {

// 释放共享读锁

sl.unlockRead(stamp);

}

}

// 返回计算结果

return Math.sqrt(currentX \* currentX + currentY \* currentY);

}

// 获取读锁，并尝试转换为写锁

void moveIfAtOrigin(double newX, double newY) {

long stamp = sl.tryOptimisticRead();

try {

// 如果当前点在原点则移动

while (x == 0.0 && y == 0.0) {

// 尝试将获取的读锁升级为写锁

long ws = sl.tryConvertToWriteLock(stamp);

// 升级成功，则更新stamp，并设置坐标值，然后退出循环

if (ws != 0L) {

stamp = ws;

x = newX;

y = newY;

break;

} else {

// 读锁升级写锁失败则释放读锁，显示获取独占写锁，然后循环重试

sl.unlockRead(stamp);

stamp = sl.writeLock();

}

}

} finally {

sl.unlock(stamp);

}

}}

1. 谈下对基于链表的非阻塞无界队列 ConcurrentLinkedQueue 原理的理解？

<https://segmentfault.com/a/1190000016248143>

1. ConcurrentLinkedQueue 内部是如何使用 CAS 非阻塞算法来保证多线程下入队出队操作的线程安全？

入队：

public boolean offer(E e) {

checkNotNull(e);

final Node<E> newNode = new Node<E>(e);

for (Node<E> t = tail, p = t;;) {

Node<E> q = p.next;

if (q == null) {

// p is last node

if (p.casNext(null, newNode)) {

// Successful CAS is the linearization point

// for e to become an element of this queue,

// and for newNode to become "live".

if (p != t) // hop two nodes at a time每两次更新一次tail

casTail(t, newNode); // Failure is OK.

return true;

}

// Lost CAS race to another thread; re-read next

}

else if (p == q)

// We have fallen off list. If tail is unchanged, it

// will also be off-list, in which case we need to

// jump to head, from which all live nodes are always

// reachable. Else the new tail is a better bet.

p = (t != (t = tail)) ? t : head;

else

// Check for tail updates after two hops.取下一个节点或最后一个节点

p = (p != t && t != (t = tail)) ? t : q;

}

}

出队：

public E poll() {

restartFromHead:

for (;;) {

for (Node<E> h = head, p = h, q;;) {

E item = p.item;

if (item != null && p.casItem(item, null)) {

// Successful CAS is the linearization point

// for item to be removed from this queue.

if (p != h) // hop two nodes at a time

updateHead(h, ((q = p.next) != null) ? q : p);

return item;

}

else if ((q = p.next) == null) {

updateHead(h, p);

return null;

}

else if (p == q)

continue restartFromHead;

else

p = q;

}

}

}

1. 基于链表的阻塞队列 LinkedBlockingQueue 原理。

LinkedBlockingQueue是一种近似有界阻塞队列，为什么说近似？因为LinkedBlockingQueue既可以在初始构造时就指定队列的容量，也可以不指定，如果不指定，那么它的容量大小默认为Integer.MAX\_VALUE。

LinkedBlockingQueue除了底层数据结构（单链表）与ArrayBlockingQueue不同外，另外一个特点就是：

它维护了两把锁——takeLock和putLock。

takeLock用于控制出队的并发，putLock用于入队的并发。这也就意味着，同一时刻，只能只有一个线程能执行入队/出队操作，其余入队/出队线程会被阻塞；但是，入队和出队之间可以并发执行，即同一时刻，可以同时有一个线程进行入队，另一个线程进行出队，这样就可以提升吞吐量。

在ArrayBlockingQueue维护了一把全局锁，无论是出队还是入队，都共用这把锁，这就导致任一时间点只有一个线程能够执行。那么对于“生产者-消费者”模式来说，意味着生产者和消费者不能并发执行。

1. 阻塞队列LinkedBlockingQueue 内部是如何使用两个独占锁 ReentrantLock 以及对应的条件变量保证多线程先入队出队操作的线程安全？
2. CountDownLatch 与线程的 Join 方法区别是什么？
3. Semaphore 的内部实现是怎样的？
4. 并发组件CopyOnWriteArrayList 是如何通过写时拷贝实现并发安全的 List？

CopyOnWriteArrayList的解决方案如下：

public boolean add(E e) {

final ReentrantLock lock = this.lock;

lock.lock();

try {

Object[] elements = getArray();

int len = elements.length;

Object[] newElements = Arrays.copyOf(elements, len + 1);

newElements[len] = e;

setArray(newElements);

return true;

} finally {

lock.unlock();

}

}

public E get(int index) {

return get(getArray(), index);

}

读读并行不做任何处理；

写写并行通过对写操作进行上锁来解决(使用锁机制ReentrantLock来串行化所有写操作)

读写并行通过对写方式的改造来解决(所有写操作完成，替换整个array内容)

ReentrantLock

//公平锁，非公平锁

public ReentrantLock(boolean fair);

//等待超时返回false

public boolean tryLock(long timeout, TimeUnit unit)

//可中断

public void lockInterruptibly() throws InterruptedException

Condition

void await() throws InterruptedException;

void awaitUninterruptibly();  
boolean awaitUntil(Date deadline) throws InterruptedException;

boolean await(long time, TimeUnit unit) throws InterruptedException;

void signal();

void signalAll();

//

await()方法会使当前线程等待，同时释放当前锁，当其他线程调用signal方法或者signalAll方法时会重新获得锁并继续执行。

awaitUninterruptibly方法和await方法一样，区别是不会响应中断

Condition必须和ReentrantLock配合一起使用

lock.lock();

...do something

condition.await();

...do something

lock.unlock();

允许多个线程同时访问：信号量Semaphone

public Semaphore(int permits) 允许多个线程

public Semaphore(int permits, boolean fair)公平非公平信号量

public void acquire() throws InterruptedException

public void acquireUninterruptibly()

ReadWriteLock读写锁

读读不互斥

读写互斥

写写互斥

1. BlockingQueue 接口介绍

接口名 介绍

add（E e） 如果队列没有空间。则抛出异常

boolean offer(E e) 没有多余空间。则返回false

void put(E e) 没有多余的空间，则进行等待

boolean offer(E e, long timeout, TimeUnit unit) 没有多余的空间，则进行等待。具有超时机制

E take() 从头部取出元素。如果没有则进行等待

E poll(long timeout, TimeUnit unit) 从头部取出元素。如果没有则进行等待。具有超时机制。

int drainTo(Collection c) 把队列中的元素移除到集合中。比单独的poll更加高效。

2. BlockingDeque 接口介绍

接口名 介绍

void addFirst(E e) 添加元素到头部。如果没有多余的空间。则扔出异常。

void addLast(E e) 添加元素到尾部。如果没有多余的空间。则抛出异常。

boolean offerFirst(E e) 添加元素到头部。如果没有多余的空间。则返回false

boolean offerLast(E e) 添加元素到尾部。如果没有多余的空间。则返回false

void putFirst(E e) 添加元素到头部。如果没有多余的空间。则进行等待

void putLast(E e) 添加元素到尾部。如果没有多余的空间。则进行等待

boolean offerFirst(E e, long timeout, TimeUnit unit) 添加元素到头部。如果没有多余的空间。则进行等待。具有超时机制

boolean offerLast(E e, long timeout, TimeUnit unit) 添加元素到尾部。如果没有多余的空间。则进行等待。具有超时机制

E takeFirst() 查询并移除头部元素。如果没有多余的空间。则进行等待。

E takeLast() 查询并移除尾部元素。如果没有多余的空间。则进行等待。

ConcurrentLinkedQueue是一个基于链表的无界非阻塞队列，并且是线程安全的

offer()：将指定的元素插入队列的尾部

poll() ：获取并移除队列的头，如果队列为空则返回null

peek()：获取表头元素但不移除队列的头，如果队列为空则返回null。

remove(Object obj)：移除队列已存在的元素，返回true，如果元素不存在，返回false。

add(E e)：将指定元素插入队列末尾，成功返回true，失败返回false（此方法非线程安全的方法，不推荐使用）。

# **CLH锁**

CLH(Craig, Landin, and Hagersten locks): 是一个自旋锁，能确保无饥饿性，提供先来先服务的公平性。

CLH锁也是一种基于链表的可扩展、高性能、公平的自旋锁，申请线程只在本地变量上自旋，它不断轮询前驱的状态，如果发现前驱释放了锁就结束自旋。

当一个线程需要获取锁时：

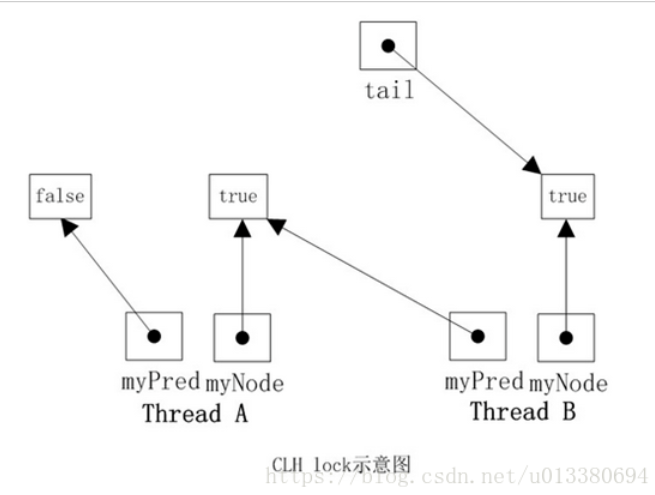
a.创建一个的QNode，将其中的locked设置为true表示需要获取锁

b.线程对tail域调用getAndSet方法，使自己成为队列的尾部，同时获取一个指向其前趋结点的引用myPred

c.该线程就在前趋结点的locked字段上旋转，直到前趋结点释放锁

d.当一个线程需要释放锁时，将当前结点的locked域设置为false，同时回收前趋结点

如下图，线程A需要获取锁，其myNode域为true，tail指向线程A的结点，然后线程B也加入到线程A后面，tail指向线程B的结点。然后线程A和B都在其myPred域上旋转，一旦它的myPred结点的locked字段变为false，它就可以获取锁。明显线程A的myPred locked域为false，此时线程A获取到了锁



public class CLHLock implements Lock {

AtomicReference<QNode> tail = new AtomicReference<QNode>(new QNode());

ThreadLocal<QNode> myPred;

ThreadLocal<QNode> myNode;

public CLHLock() {

tail = new AtomicReference<QNode>(new QNode());

myNode = new ThreadLocal<QNode>() {

protected QNode initialValue() {

return new QNode();

}

};

myPred = new ThreadLocal<QNode>() {

protected QNode initialValue() {

return null;

}

};

}

@Override

public void lock() {

QNode qnode = myNode.get();

qnode.locked = true;

QNode pred = tail.getAndSet(qnode);

myPred.set(pred);

while (pred.locked) {

}

}

@Override

public void unlock() {

QNode qnode = myNode.get();

qnode.locked = false;

myNode.set(myPred.get());

}

}