进程、线程、管程、协程
程序计数器私有?
虚拟机栈和本地方法栈私有?
堆和方法区
逃逸分析
使用多线程可能带来的问题：内存泄漏、死锁、线程不安全
线程的阻塞状态和等待状态的区别
死锁
活锁
线程的创建方式
Synchronized关键字
对象头格式
synchronized轻量级锁
锁膨胀
自旋优化
偏向锁
撤销偏向锁
批量重偏向
批量撤销
锁消除
java的锁
AQS
ReentrantLock(可重入锁)
ReentrantLock非公平锁加锁
ReentrantLock非公平锁释放锁
ReentrantLock公平锁
Synchronized和ReentrantLock的区别
读写锁ReentrantReadWriteLock
读写锁StampedLock
Semaphore信号量
Volatile
volatile底层原理
并发的三大特性
Synchronized和volatile的区别
ThreadLocal
在ThreadLocal.get()操作后，GC后key是否为null?
ThreadLocal的Hash
ThreadLocal的扩容
ThreadLocal内存泄漏的问题
线程池ThreadPoolExecutor
ThreadPoolExecutor构造参数
ThreadPoolExecutor拒绝策略
各种线程池
线程池提交方法
关闭线程池
tomcat线程池
Fork/Join线程池
CAS比较并交换
Atomic原子类
AtomicInteger的原理
LongAdder原子累加器
Unsafe
sleep和wait
wait/notify
park/unpark原理

进程、线程、管程、协程

进程：程序的一次执行过程，是动态的，是资源分配的基本单位。

线程：轻量级进程，线程在进程内部活动，一个进程可以有多个线程，java中线程共用进程的堆空间和方法区，每个线程又有自己的程序计数器、虚拟机栈和本地方法栈。线程间的切换比进程间的切换要小很多。

协程：是一种用户态的轻量级线程，协程的调度由用户控制。

管程：用来管理进程的，定义一种数据结构和控制一些进程操作的集合。synchronized底层的加锁就是使用的管程monitor。

程序计数器私有?

程序计数器主要做两件事：
1.字节码解释器通过改变程序计数器来读取指令，实现代码的流程控制；
2.在多线程中，程序计数器主要记录当前线程的执行位置，保证线程被切换回来的时候能知道上次运行到什么位置了。
所以程序计数器私有化主要是为了保证线程切换后能恢复到正常的执行位置。

虚拟机栈和本地方法栈私有?

虚拟机栈：存放的是栈帧。每个java方法在执行的时候会创建一个栈帧放入虚拟机栈中。这个栈帧中保存了局部变量表、操作数栈、常量池引用等信息。(局部变量表中放方法中的参数和变量，还有对象的引用)
本地方法栈：和虚拟机栈作用相似。主要是存放native方法的数据。
私有化是为了保证线程中的局部变量不会被其他线程访问。

堆和方法区

堆和方法区是所有线程共享的资源，堆是进程中最大的一块内存，主要用于存放新创建的对象实例，几乎所有对象都在这里分配内存。方法区是存放已经加载了的类信息、常量、静态变量等数据。
对象实例数据：放在堆中，指对象中各个实例字段的数据；
对象类型数据：放在方法区，表示对象的类型、父类、实现的接口、方法等。

所有对象一定会在堆中创建吗？
不一定，通过逃逸分析，如果对象不发生逃逸，且在方法中创建对象，在方法中销毁，那么这个对象可以在栈上进行分配。这样做的好处是减少垃圾回收次数，提高性能。

逃逸分析

目的：减少堆内存的分配压力

如果一个对象在方法中被定义，还可能被外部的方法所引用，则称为方法逃逸。

```
public static StringBuffer craeteStringBuffer(String s1, String s2) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
```

```
    sb.append(s2);
    return sb;
}
```

这个StringBuffer sb是一个局部变量，但是被作为返回值返回，因此可能在其他方法中被引用，称这个变量逃逸到了方法外部。

不想逃逸到外部的写法：

```
public static String createStringBuffer(String s1, String s2) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    return sb.toString();
}
```

如果使用了逃逸分析，则编译器可以对代码进行优化：

1. 同步省略：

   如果一个对象被发现只能从一个线程里被访问，则对于这个对象的操作可以不考虑同步。

   ```
   public void f() {
       Object hollis = new Object();
       synchronized(hollis) {
           System.out.println(hollis);
       }
   }
   ```

   如果hollis对象只在该方法中，所以并不会被其他线程所访问，因此JIT在编译阶段就会优化代码，等价于：

   ```
   public void f() {
       Object hollis = new Object();
       System.out.println(hollis);
   }
   ```

2. 将堆分配转化为栈分配

   如果一个对象在子程序中被分配，要使指向该对象的指针不会逃逸，则可能会在栈中进行分配内存。在栈上分配内存，就无需垃圾回收

3. 分离对象或标量替换

   有的对象可能不需要作为一个连续的内存结构存在也可以被访问到，那么对象的部分(或全部)可以不存储在内存中，而是存储在CPU寄存器中。

使用多线程可能带来的问题：内存泄漏、死锁、线程不安全

内存泄漏就是堆内存中一些不使用的对象，但是却无法被回收，这样就一直在内存中，最终导致内存泄露。

例子：

大量使用static、常量之类的；

创建连接、打开流，结束后没有关闭；

内存泄露检测：使用工具(Java VisualVM)

线程的阻塞状态和等待状态的区别

> 两者都表示当前线程的暂停执行，但是进入等待状态的线程是自己主动发起等待的，而阻塞是被动的。
>
> 阻塞发生在同步代码块外面，比如获取不到锁而被阻塞；
> 等待是在自己进入的，比如Object.wait().

死锁

为了增强并发度，一般会设置多把细粒度的锁。但是当一个线程需要同时获取多把锁的时候，就容易产生死锁。

```
//t1线程获得A对象的锁，然后想获取B对象的锁
//t2线程获得B对象的锁，然后想获取A对象的锁
Object A = new Object();
Object B = new Object();

new Thread(() -> {
    synchronized (A){
        log.debug("lock A");
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (B){
            log.debug("lock B");
            log.debug("操作....");
        }
    }
}, "t1").start();

new Thread(() -> {
    synchronized (B){
        log.debug("lock B");
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        synchronized (A){
            log.debug("lock A");
            log.debug("操作....");
        }
    }
}, "t2").start();
```

死锁的必要条件：
    请求和保持；
    互斥；
    不可剥夺；
    循环等待；
预防死锁：
    破坏请求和保持条件：一次性申请所有需要的资源；
    破坏不可剥夺条件：如果申请不到资源，则主动释放；
    破坏循环等待条件：按顺序申请资源。线程1先获取A对象的锁，再获取B对象的锁；线程2也是一样。但是这样顺序加锁容易造成饥饿的问题

活锁

与死锁恰好相反，两个线程没有被阻塞，但是由于双方在不断改变对方的结束条件，导致两个线程永远无法结束

线程的创建方式

```
//1.直接new Thread()，并重写run方法
Thread t = new Thread(){
    @Override
    public void run() {
        log.debug("running");
    }
};

//2. 通过Runnable创建，并重写run方法
Runnable runnable = new Runnable(){
    @Override
    public void run() {
        log.debug("running");
    }
};
Thread t2 = new Thread(runnable, "t2");

//3.通过FutureTask，支持返回值
FutureTask<Integer> task = new FutureTask<>(new Callable<Integer>() {
    @Override
    public Integer call() throws Exception {
        log.debug("running.....");
        Thread.sleep(2000);
        return 100;
    }
});
Thread t3 = new Thread(task, "t3");
t3.start();

runnable和callable：
    runnable不能返回结果以及无法抛出检查异常；
    callable都可以实现。
```

Synchronized关键字

synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只有一个线程在执行。早期的java版本中synchronized属于重量级锁，效率很低。

为什么早期效率低？

因为早期技术不成熟，过于依赖操作系统底层的锁来实现。在切换线程时需要从用户态到内核态，这里需要较长的时间，所以效率低。

在jdk1.6之后，对synchronized有了很大的优化，引入无锁、偏向锁、轻量锁、重量锁来对synchronized分级，大大提升效率。

synchronized用法：

```
//修饰实例方法：对当前对象实例加锁，也等价于给对象加锁
public synchronized void increment(){              //synchronized只能锁对象，即使是加在方法上，也是锁的对象
        count++;
    }

public void decrement(){                           //等价于上面那种写法
    synchronized (this){
        count--;
    }
}

/*
    修饰静态方法(也就是修饰类对象)：对当前类对象加锁。如果一个线程 A 调用一个实例对象的非静态 synchronized 方法，而线程 B 需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁。
*/
public synchronized static void test(){           //synchronized加在静态方法上面，锁的是类对象，等价于下面那种写法

    }

public static void test1(){
    synchronized (Room.class){

    }
}

//修饰代码块：给指定对象/类加锁
public int getCount(){
    synchronized (this){
        return count;
    }
}
```

synchronized关键字的底层原理：

修饰代码块：

monitorenter和monitorexit来对同步代码块进行加锁。

修饰方法：

有一个ACC_SYNCHRONIZED标识，标识该方法为同步方法。

对象头格式

Mark Word (64 bits)					State
unused:25	hashcode:31	unused:1	age:4	biased_lock:0	01	Normal
thread:54	epoch:2	unused:1	age:4	biased_lock:1	01	Biased
ptr_to_lock_record:62					00	Lightweight Locked
ptr_to_heavyweight_monitor:62					10	Heavyweight Locked
					11	Marked for GC

synchronized轻量级锁

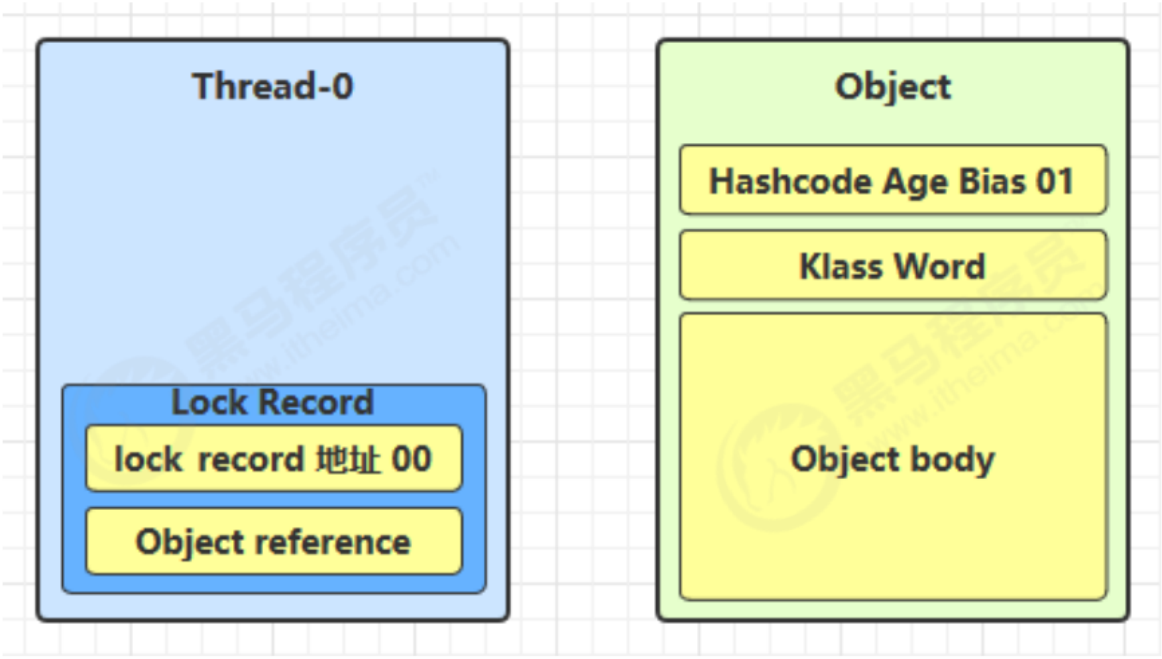
轻量级锁的使用场景：如果一个对象虽然有多个线程要对它加锁，但是加锁的时间是错开的(没有竞争)，则可以使用轻量级锁来优化

轻量级锁对使用者来说是透明的，直接使用synchronized即可

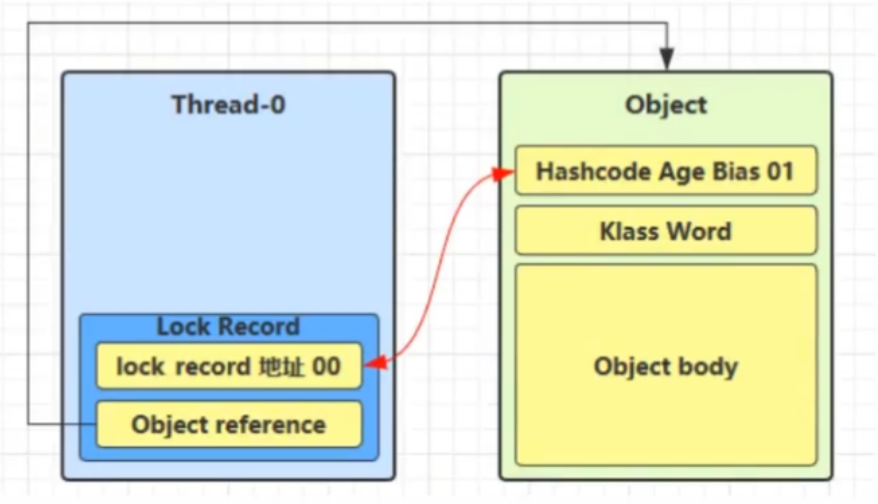
```
static final Object obj = new Object();
public static void method1() {
    synchronized( obj ) {
        // 同步块 A
        method2();
    }
}
public static void method2() {
    synchronized( obj ) {
        // 同步块 B
    }
}
```

首先执行到method1方法的synchronized代码块：

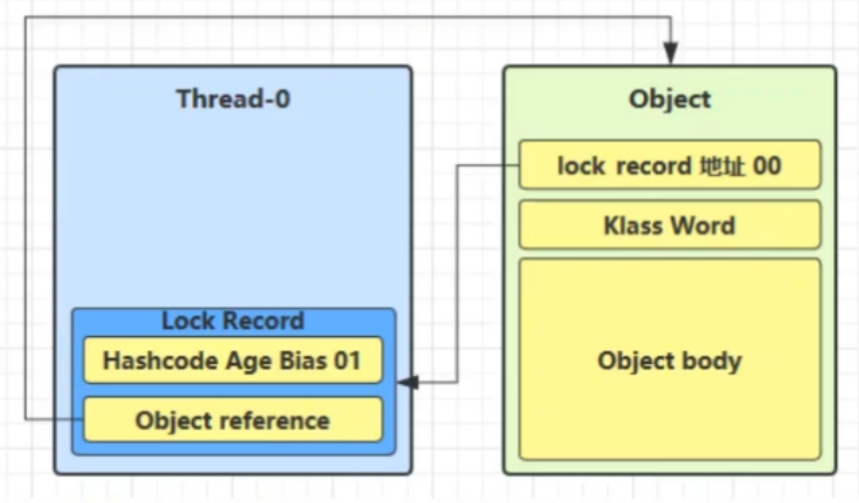
该线程会在栈帧中创建一个锁记录Lock Record对象，该记录里面可以存放对象引用地址和要加锁对象的Mark Word(对象头中的字段)



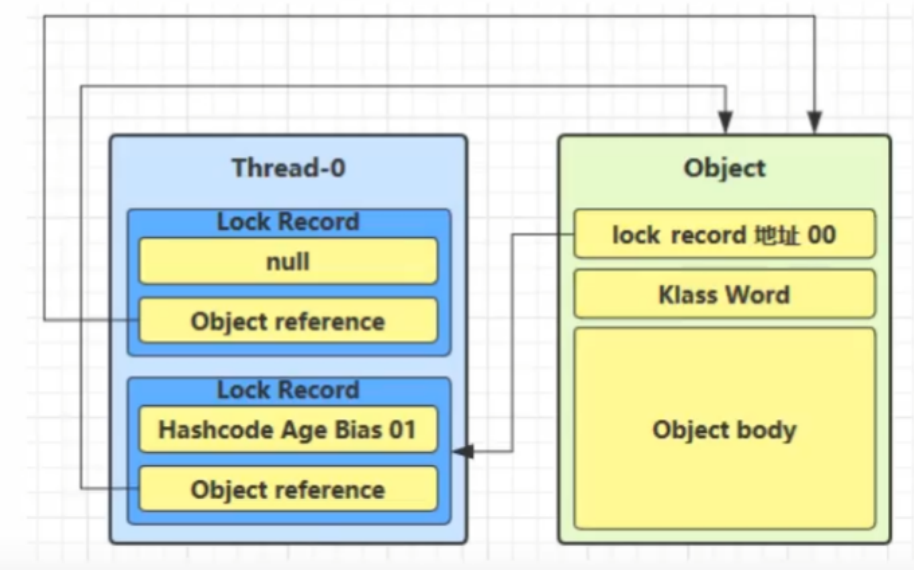
锁记录中的Object reference指向加锁对象，并尝试使用CAS来替换加锁对象Object中的Mark Word，将Mark Word中的值(哈希、分代年龄、锁状态)存入锁记录中。**01是无锁状态**，则可以被CAS替换，**轻量级锁是00。**(注意只有状态为01即无锁状态才会交换成功)



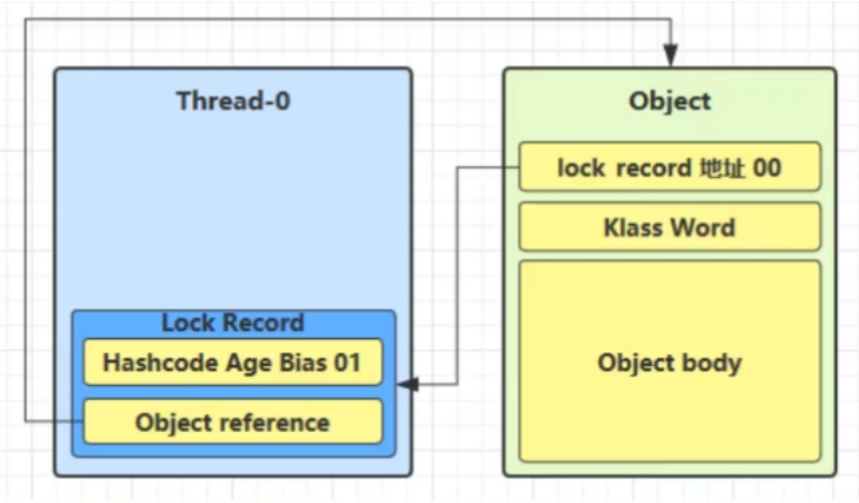
如果CAS替换成功，则表示加锁成功，对象头中存储锁记录地址和状态00



当执行到method2方法时，又要对同一个对象加锁。但是这个对象已经被该线程加锁，因此会CAS失败，但是该线程还是会在栈帧中添加一个新的Lock Record作为synchronized锁重入的计数



method2方法执行完毕，发现Lock Record中的值为null，则判断为锁重入，清除这个Lock Record，表示锁计数减一

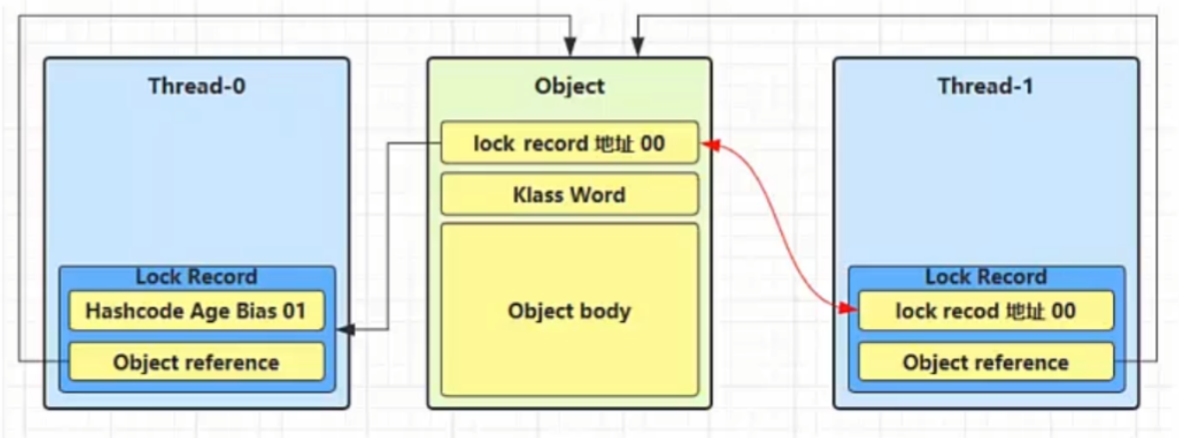


如果Lock Record中的值不为null，则使用CAS将Mark Word的值恢复给对象头，如果失败，则表示轻量级锁进行了锁膨胀或已经升级为重量级锁，则进入重量级锁的解锁流程。

锁膨胀

如果在进行CAS操作的时候发现对象头中的Mark Word无法交换，则表示有其他线程已经为该对象加上了轻量级锁(有竞争)，这个时候就需要进行锁膨胀，将轻量级锁变为重量级锁。

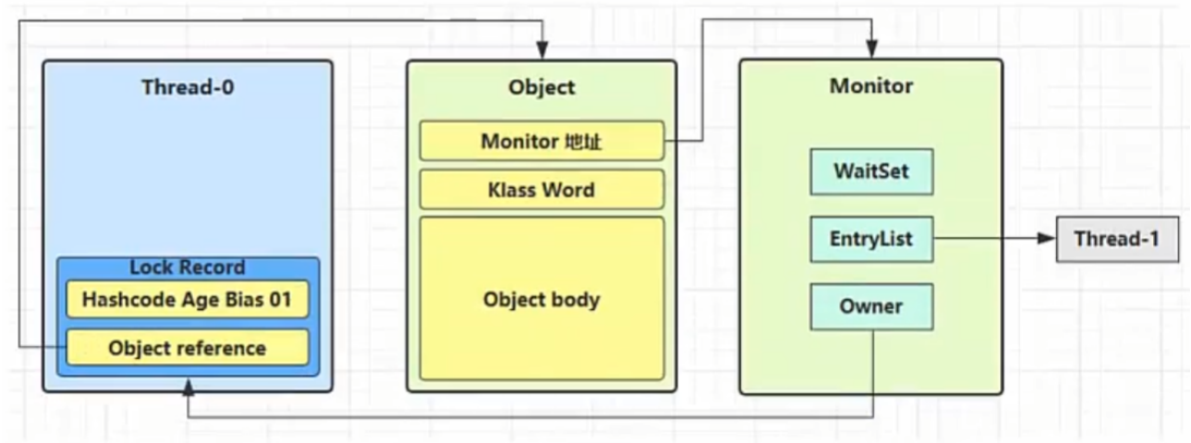
当Thread1进行轻量级加锁时，Thread0已经为这个对象加上了轻量级锁。



Thread1进入锁膨胀流程：

为该对象申请Monitor锁，让该对象指向重量级锁的地址，并且锁状态变为**10(重量级锁)**；

自己进行Monitor的EntryList BLOCKED队列。



当Thread0执行完毕synchronized代码块的时候，使用CAS将Mark Word的值恢复给对象头，由于对象头记录的已经不是轻量级锁的信息，因此会失败。这时会进入重量级锁的解锁流程：

按照Monitor地址找到Monitor对象；

设置Owner为null；

唤醒EntryList中的BLOCKED线程。

> Monitor中的WaitSet是为正在执行线程调用了wait方法后保留的，执行wait方法后，执行的线程会进入WaitSet，当调用notify方法后，该线程会从waitSet转移到EntryList中阻塞，等待获取锁。

自旋优化

在重量级锁竞争的时候，如果当前线程自旋成功(持锁线程完成同步块代码，释放了锁)，这时当前线程就避免了阻塞，减少开销。自旋就是不进入阻塞状态，一直重试看能否获得锁。

自旋会占用CPU时间，所以只有多核CPU使用自旋才会有效。

偏向锁

轻量级锁在没有竞争的时候，如果有重入的情况，每次还是要进行CAS操作。因此Java6引入偏向锁来对其进行优化。

只有第一次使用CAS的时候，会将线程ID设置到对象头的Mark Word中，之后如果发现这个线程ID属于自己就表示没有竞争，则不用CAS。

对象创建后，对象头中的Mark Word的值为0x05，即最后三位是101，这是对象头中的thread、epoch、age都是0

偏向锁是延迟的，不会在程序启动的时候立即生效，如果想避免延迟，则可以加JVM参数：-xx:BiasedLockingStartupDelay=0来禁用延迟

撤销偏向锁

> 1. 调用对象的hashCode方法。因为如果是偏向锁，则Mark Word里面不会有hashCode字段。只有无锁状态才会有hashCode字段。
> 2. 如果当前线程对某个对象加锁，则该锁为偏向锁，但是之后又有另外一个线程需要对该对象加锁，则该对象的偏向锁会升级为轻量级锁。
> 3. 调用wait/notify也会撤销偏向锁，因为wait/notify只有重量级锁才有。

批量重偏向

如果对象虽然被多个线程访问，但是没有竞争，这时偏向T1线程的对象也是有机会偏向T2的，重偏向会重置对象的Thread ID。

重偏向的时机：当撤销偏向锁次数超过20次的时候(**这个次数并不是对单一对象来说的，而是在jvm中任意对象只要发生撤销偏向锁，即增加一次**)，jvm会在给对象加锁时重新偏向至加锁线程。

批量撤销

如果撤销偏向锁的次数超过40次，则jvm会认为偏向错了，于是整个类的所有对象都会变为不可偏向的，新创建的对象也是不可偏向的。

锁消除

可以参考逃逸分析里面的同步省略

如果一个对象被发现只能从一个线程里被访问，则对于这个对象的操作可以不考虑同步。

```
public void f() {
    Object hollis = new Object();
    synchronized(hollis) {
        System.out.println(hollis);
    }
}

如果hollis对象只在该方法中，所以并不会被其他线程所访问，因此JIT在编译阶段就会优化代码，等价于：
public void f() {
    Object hollis = new Object();
    System.out.println(hollis);
}
```

java的锁

```
乐观锁和悲观锁：
    对于同一个数据的并发操作，悲观锁认为自己在使用数据的时候一定有别的线程来修改数据，因此在获取数据的时候会先加锁，确保数据不会被别的线程修改。Java中，synchronized关键字和Lock的实现类都是悲观锁。
    乐观锁认为自己在使用数据时不会有别的线程修改数据，所以不会添加锁，只是在更新数据的时候去判断之前有没有别的线程更新了这个数据。如果这个数据没有被更新，当前线程将自己修改的数据成功写入。实现乐观锁的类AtomicInteger。
    * 悲观锁适合写操作多的场景，先加锁可以保证写操作时数据正确。
    * 乐观锁适合读操作多的场景，不加锁的特点能够使其读操作的性能大幅提升。

自旋锁和适应性自旋锁：
    在许多场景中，同步资源的锁定时间很短，为了这一小段时间去切换线程，线程挂起和恢复现场的花费可能会让系统得不偿失。如果物理机器有多个处理器，能够让两个或以上的线程同时并行执行，我们就可以让后面那个请求锁的线程不放弃CPU的执行时间，看看持有锁的线程是否很快就会释放锁。而为了让当前线程“稍等一下”，我们需让当前线程进行自旋，如果在自旋完成后前面锁定同步资源的线程已经释放了锁，那么当前线程就可以不必阻塞而是直接获取同步资源，从而避免切换线程的开销。这就是自旋锁。

无锁、偏向锁、轻量级锁、重量级锁：专门针对于synchronized。锁的状态只能升不能降。
    重量级锁：依赖于操作系统的Mutex Lock（互斥锁）来实现的锁。效率很低。JDK6之前synchronized的实现方式。

    无锁：没有对资源进行锁定，所有线程都能访问同一个资源，但是同时只有一个线程能修改成功。
    偏向锁：一段同步代码一直被一个线程所访问，则该线程会自动获取锁，降低锁的获取代价。
```

轻量级锁：当锁是偏向锁的时候，如果被另外的线程所访问，则偏向锁升级为轻量级锁，其他线程通过自旋尝试获取锁。

重量级锁：若当前只有一个等待线程，则该线程通过自旋进行等待。但是当自旋超过一定的次数或者一个线程持有锁，一个在自旋，又有第三个线程来访问时，轻量级锁升级为重量级锁。此时等待的线程都会进入阻塞状态。

公平锁和非公平锁：

公平锁：多个线程按照申请锁的顺序来获取锁，线程直接进入队列中排队，队列中的第一个线程可以获得锁。优点是等待锁的线程不会被饿死。缺点是整体吞吐效率比非公平锁要低，等待队列中除第一个线程外所有线程都会被阻塞。CPU唤醒阻塞线程开销比非公平锁大。

非公平锁：多个线程加锁时直接尝试获取锁，获取不到才会加入等待队列。但如果此时锁刚好可用，则这个线程可以直接获取到锁。优点：减少唤起线程的开销，整体吞吐率高。缺点：等待队列中的线程可能会饿死，或者很久才会获得锁。

可重入锁和非可重入锁：

可重入锁（递归锁）：同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁，不会因为之前已经获取过还没释放而阻塞。优点：一定程度可以避免死锁。

非可重入锁：当前线程执行某个方法获取了锁，在这个方法中再次尝试获取锁时，会因为无法获取而阻塞。产生死锁。

独享锁和共享锁：都是悲观锁

独享锁（排他锁）：该锁一次只能被一个线程所持有。如果一个线程对数据A加上排他锁，则其他线程无法对数据A加任何类型的锁。获得排他锁的线程可以读写数据。

共享锁：该锁可以被多个线程所持有。如果线程T对数据A加上共享锁后，其他线程只能对数据A加共享锁，不能加排他锁。共享锁的线程只能读数据，无法修改数据。

AQS

是阻塞式锁和相关同步器工具的框架

使用state属性(volatile修饰)来表示资源的状态(独占or共享)，子类需要定义如何维护这个状态，控制如何获取锁和释放锁

- getState：获取state状态
- setState：设置state状态
- compareAndSetState：通过cas机制设置state状态(只是借鉴cas机制，本身还是阻塞式的锁)
- Exclusive(独占状态)：只有一个线程能执行。比如ReentrantLock，又分为公平锁和非公平锁。
- Share(共享状态)：多个线程可以同时执行

提供了基于FIFO的等待队列，类似于Monitor的EtrySet
条件变量来实现等待、唤醒机制，支持多个条件变量。类似于Monitor的WaitSet

ReentrantLock(可重入锁)

```
//基本语法
ReentrantLock reentrantLock = new ReentrantLock();
reentrantLock.lock();
try{
    //临界区
}finally{
    reentrantLock.unlock();
}
```

可重入实现原理：

在线程已经获得锁后，如果判断获取锁的线程还是该线程，则表示发生了锁重入，将state++

可打断特性，打断获取锁失败，进入等待队列的线程，中止等待

```
Thread t1 = new Thread(() -> {
    try {
        /*
          如果没有竞争，lockInterruptibly()方法就会获取lock对象锁
          如果有竞争，则进入阻塞队列等待，但是这个等待可以由其他线程执行interrupt方法中止等待
        */
        log.debug("尝试获取锁.....");
        lock.lockInterruptibly();
    } catch (InterruptedException e) {
        e.printStackTrace();
        log.debug("没有获取到锁....");
        return;
    }
    try {
        log.debug("获取到锁....");
    }finally {
        lock.unlock();
    }
}, "t1");

lock.lock();
t1.start();

Thread.sleep(1000);
log.debug("打断t1线程的等待......");
t1.interrupt();     //这里是主线程执行的
```

锁超时，在获取锁的时候，如果一直在等待，且超过设置时间，则自动停止等待

```
Thread t1 = new Thread(() -> {
    log.debug("尝试获取锁.....");
    try {
        if (!lock.tryLock(1, TimeUnit.SECONDS)) {       //尝试等待1秒，如果超过1秒还
没有释放锁，则返回false
            log.debug("获取锁失败....");
            return;
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
        log.debug("获取锁失败....");
        return;
    }
    try {
        log.debug("获取锁成功，执行临界区代码....");
    }finally {
        lock.unlock();
    }
}, "t1");

lock.lock();
log.debug("获得到锁.....");
t1.start();

Thread.sleep(500);
log.debug("释放了锁.....");
lock.unlock();
```

公平性：Synchronized和ReentrantLock都默认是非公平的，但是ReentrantLock可以设置为公平锁，先进入等待队列的在锁释放的时候先获取。但是公平锁会降低并发度。所以一般不使用。

条件变量：

```
synchronized也有条件变量，当不满足条件的时候会进入WaitSet等待
ReentrantLock的条件变量相比于synchronized来说就是支持多个条件变量
    相当于ReentrantLock可以有多个WaitSet对应不同的条件

//创建一个新的条件变量
Condition condition1 = lock.newCondition();
Condition condition2 = lock.newCondition();

lock.lock();

//进入条件1的等待，await和wait类似
/*
    await之前需要先获取锁；
    await执行后会释放锁，进入对应的条件等待队列；
    await被唤醒(打断或者超时)后，会重新竞争lock锁；
    竞争成功后await后面的代码才会继续执行
*/
condition1.await();

//唤醒条件1中的某个线程
condition1.signal();

//唤醒条件1中的全部线程
condition1.signalAll();
```

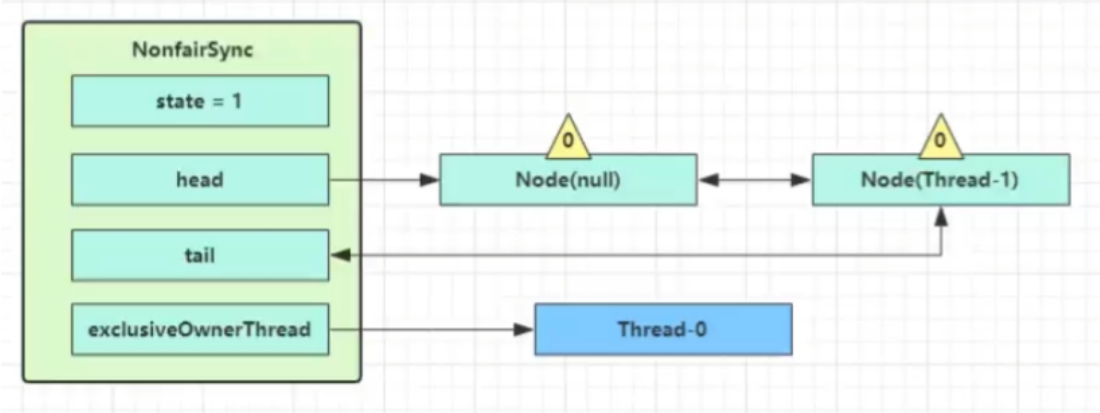
ReentrantLock非公平锁加锁

```
//ReentrantLock默认就是非公平锁
public ReentrantLock(){
    sync = new NofairSync();
}

static final class NonfairSync extends Sync {
    private static final long serialVersionUID = 7316153563782823691L;

    /**
         * Performs lock.  Try immediate barge, backing up to normal
         * acquire on failure.
         */
    final void lock() {
        if (compareAndSetState(0, 1))
            setExclusiveOwnerThread(Thread.currentThread());
        else
            acquire(1);
    }

    protected final boolean tryAcquire(int acquires) {
        return nonfairTryAcquire(acquires);
    }
}

public final void acquire(int arg) {
```

```
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

只有一个线程，则会正常加锁，如果又来了一个线程来竞争：

1. CAS尝试将state由0改为1，失败
2. 进入tryAcquire逻辑，再次尝试将state由0改为1，失败
3. 进入addWaiter逻辑，构造Node队列：



3.1 黄色三角形表示该Node的waitStatus状态，0是正常状态

3.2 Node的创建时懒惰的(到了这个逻辑才会创建)

3.3 第一个Node称为Dummy(哑元)或哨兵，只用来占位，没有关联线程

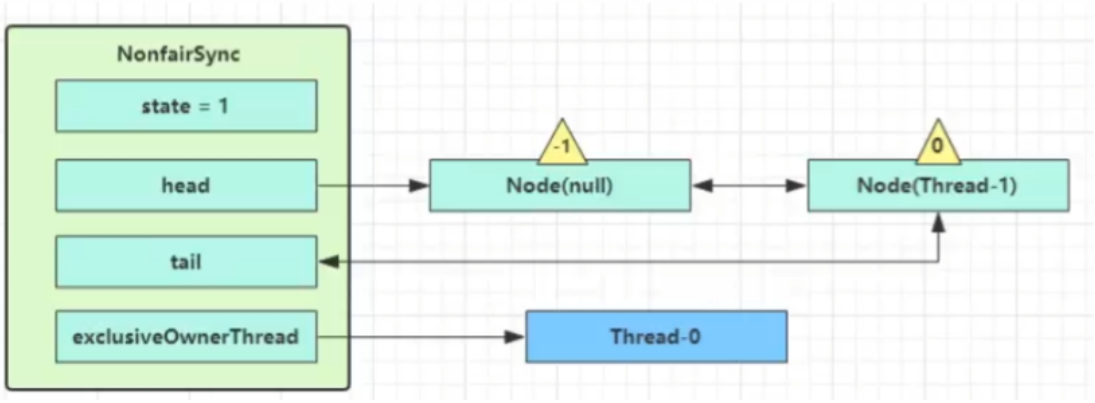
```
final boolean acquireQueued(final Node node, int arg) {
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                p.next = null; // help GC
                failed = false;
                return interrupted;
            }
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

4. 进入acquireQueue逻辑

4.1 acquireQueue会在一个死循环中不断尝试获取锁。如果还是失败则进入park阻塞

4.2 如果自己紧挨着head(排在第二位)，那么再次tryAcquire尝试获取锁，失败

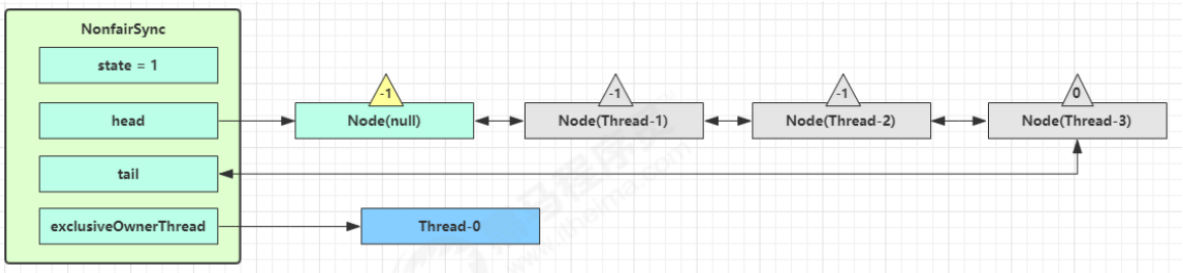
4.3 进入shouldParkAfterFailedAcquire逻辑，将前驱Node，即head的waitStatus改为-1，返回false。-1表示该节点需要唤醒后继节点



4.4 当再次进入shouldParkAfterFailedAcquire的时候，前驱的waitStatus已经是-1了，则返回true

4.5 则进入parkAndCheckInterrupt，阻塞当前线程

多个线程竞争失败后



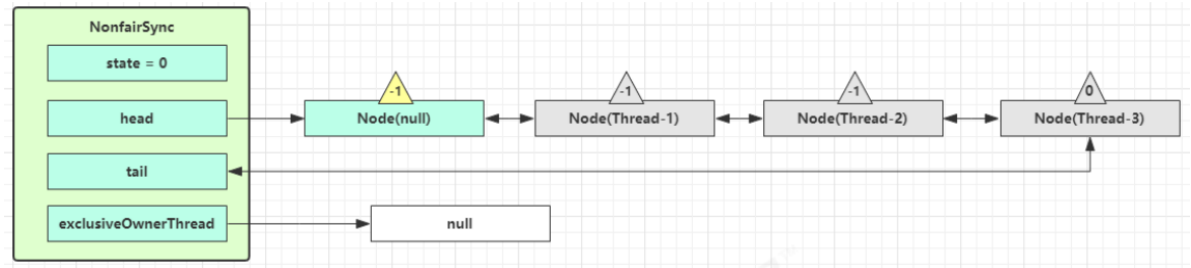
ReentrantLock非公平锁释放锁

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}

protected final boolean tryRelease(int releases) {
    int c = getState() - releases;
    if (Thread.currentThread() != getExclusiveOwnerThread())
        throw new IllegalMonitorStateException();
    boolean free = false;
    if (c == 0) {
        free = true;
        setExclusiveOwnerThread(null);
    }
    setState(c);
    return free;
}
```

1. 进入tryRelease流程，如果成功：

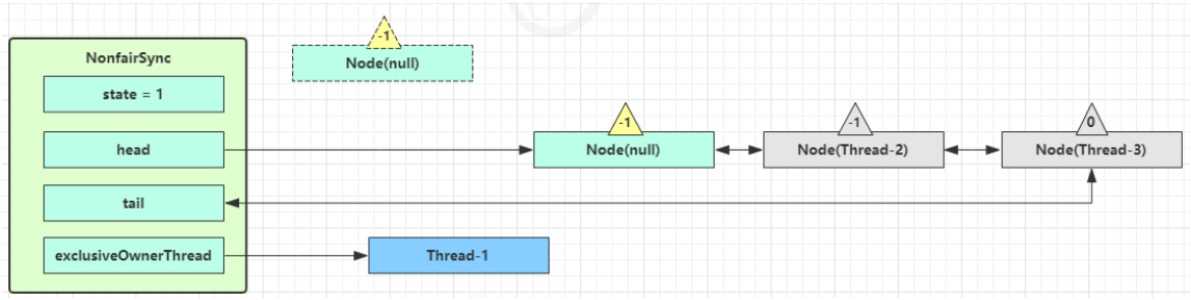
    1.1 设置exclusiveOwnerThread为null

    1.2 state设置为 0



1.3 判断head节点为不为空，以及waitStatus是否等于-1。如果满足则应该唤醒下一个节点，进入unparkSuccessor(h)流程

2. 找到队列中离head最近的一个Node，通过unpark恢复运行，然后回到唤醒线程的acquireQueued流程



3. 但是这是非公平锁。如果在唤醒之后在尝试加锁的时候又有新的线程进来，并抢先加锁，则该线程又会被阻塞

ReentrantLock公平锁

和非公平锁的主要区别在于：如果state=0，在将state通过cas改为1的时候会先去判断AQS队列中是否有前驱节点

Synchronized和ReentrantLock的区别

两者都是可重入锁；
synchronized是jvm自带的，而ReentrantLock实现了Lock接口，但是ReentrantLock需要lock()和unlock()来加锁解锁；
ReentrantLock比synchronized增加了更多的功能：
    等待可中断：可以让正在等待的线程放弃等待；
    可实现公平锁(防止饥饿)：ReentrantLock可以指定公平锁还是非公平锁；Synchronized只能是非公平锁；
    增加条件变量，可以有多个等待队列。

读写锁ReentrantReadWriteLock

读写使用不同的锁来保护，主要是为了提高读取操作的性能。当只有读的时候，允许并发。但是有写操作的时候还是需要互斥

```
class DataContainer{
    private Object data;
    private ReentrantReadWriteLock lock = new ReentrantReadWriteLock();
    private ReentrantReadWriteLock.ReadLock r = lock.readLock();
    private ReentrantReadWriteLock.WriteLock w = lock.writeLock();

    public void read(){
        log.debug("获取读锁....");
        r.lock();
        try {
```

```
            log.debug("读取....");
            Thread.sleep(1000);
            //return data;
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            log.debug("释放读锁....");
            r.unlock();
        }
    }

    public void write(){
        log.debug("获取写锁....");
        w.lock();
        try {
            log.debug("写入....");
        } finally {
            log.debug("释放写锁....");
            w.unlock();
        }
    }
}
```

注意：

- 读锁不支持条件变量；
- 持有读锁的情况下去获取写锁会导致获取写锁永久等待；
- 持有写锁的情况下获取读锁允许；

原理：

读写锁用的是同一个AQS同步器，因此等待队列和state共用

区分读写锁的标志是state：

state的低16位表示的是写锁；

state的高16位表示的是读锁。

其他地方和ReentrantLock类似，读锁的时候多个线程加锁，相当于可重入，将state高16位+1.

如果唤醒的是读锁，会将等待队列中连续的读锁都给唤醒，直到下一个是写锁。

读写锁StampedLock

jdk8加入，进一步优化读的性能。特点是在使用读锁和写锁的时候必须配合【戳】的使用。使用ReentrantReadLock的时候每次都要通过cas修改状态.

```
//加读锁、解读锁
long stamp = lock.readLock();
lock.unlockRead(stamp);

//加写锁、解写锁
long stamp = lock.writeLock();
lock.unlockWrite(stamp);
```

乐观读：

StampedLock支持tryOptimistaticRead()方法(乐观读)，读取完毕后要做一次戳校验，如果校验通过，则表示这期间没有写操作，数据可以安全使用；如果校验失败，则需要重新获取读锁，保证数据的安全。

缺点：

StampedLock不支持条件变量；

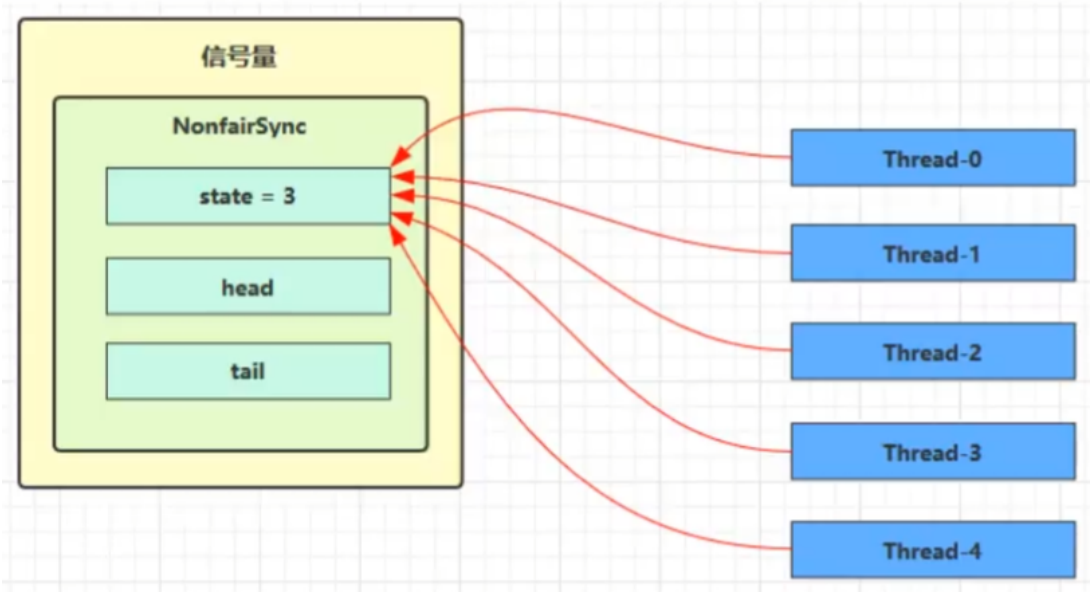
StampedLock不支持可重入。

Semaphore信号量

可以用来限制同时访问共享资源的线程上限

```
public static void test(){

    /*
       创建Semaphore对象
           支持两个参数：
               1.许可数，表示访问共享资源最大线程数
               2.公平锁，类似于ReentrantLock里面的公平和非公平
    */
    Semaphore semaphore = new Semaphore(3);

    //10个线程同时运行
    for (int i = 0; i < 10; i++) {
        new Thread(() -> {
            try {
                //获取Semaphore
                semaphore.acquire();

                log.debug("Thread running.....");
                Thread.sleep(1000);
                log.debug("Thread end.....");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }finally {
                //释放Semaphore
                semaphore.release();
            }
        }, "t" + i).start();
    }
}
```

原理，也是AQS的类似原理



Volatile

```
/*
    只能保证可见性和有序性，无法保证原子性。
    如果主存中有一个共享变量，两个线程都去访问它，为了提高速度，线程一般把变量保存在本地内存中，这样就可能造成一个线程修改主存中的值，另外一个线程还没来得及去读取主存中的值的情况，造成数据不一致。
    因此使用volatile，每次都在主存中读取变量值，同时防止JVM的指令重排(双加锁的单例模式)。
*/
public final class Singleton {

    private Singleton() { }

    private volatile static Singleton INSTANCE = null;

    public static Singleton getInstance(){
      //这个判断没有受到synchronized代码块的保护，在new Singleton();如果发生指令重排，会先赋值再调用构造函数创建对象，这时先赋值为空，会产生空指针异常
        if(INSTANCE == null){
            synchronized (Singleton.class){     //如果把synchronized加到方法上，则每次获取都要加锁，性能下降
                if(INSTANCE == null){
                    INSTANCE = new Singleton();
                }
            }
        }
        return INSTANCE;
    }
}
```

volatile底层原理

底层是内存屏障。

- 对volatile变量的**写指令后**会加入**写屏障**;
- 对volatile变量的**读指令前**会加入**读屏障**;

保证可见性

写屏障保证在该屏障之前对共享变量的改动都同步到主存中

读屏障保证在该屏障之后对共享变量的读取加载的是主存中的最新数据

保证有序性

写屏障会确保指令重排时不会将写屏障之前的代码排在写屏障之后

读屏障会确保指令重排时不会将读屏障之后的代码排在读屏障之前

并发的三大特性

原子性：一系列操作要么全部执行，要么都不执行。synchronized可以保证。
可见性：一个线程对共享变量进行了修改，另外的线程可以立即看到最新修改的值。volatile、synchronized可以保证。
有序性：代码按顺序执行。volatile、synchronized可以保证。

虽然synchronized可以保证有序性，但是在单线程下，指令重排是没有问题的。

volatile为什么不能保证原子性？
　　比如变量i被volatile修饰，虽然可以保证每次读取都是最新的值，但是如果对这个变量做了非原子的操作，那么就不能保证原子性了。比如i++。

Synchronized和volatile的区别

volatile是线程同步的轻量级实现，性能比synchronized好；
volatile能保证可见性，但是不能保证数据的原子性；synchronized都可以保证；
volatile只能用于变量，synchronized可以修饰方法和代码块。

ThreadLocal

ThreadLocal让每个线程都有自己私有的数据。如果创建了一个ThreadLocal变量，则每一个访问这个变量的线程都会有该变量的副本。

ThreadLocal的底层：
　　底层是ThreadLocalMap，是一个HashMap，但是这个map并没有链表结构。
　　可以调用get()方法：获取默认值；
　　可以调用set()方法：将默认值修改为当前线程中副本的值。在ThreadLocalMap中存放key=当前线程，value=要设置的值

在ThreadLocal.get()操作后，GC后key是否为null?

ThreadLocal的key是弱引用，如果不做操作，那么在发生GC的时候就会被回收，key变为null；
但是有get()操作就表示强引用还是存在的，因此key并不为null。

ThreadLocal的Hash

ThreadLocal中有一个属性是HASH_INCREMENT = 0X61C88647
每创建一个ThreadLocal下一个ThreadLocal的哈希就会增加一个HASH_INCREMENT

这个数叫做黄金分割数，好处是使得hash分布很均匀。

解决hash冲突的方法：

因为没有了链表，所以无法使用拉链法解决哈希冲突。这里使用的是线性探测法。由于key就相当于一个ThreadLocal，因此如果得到key值一样的情况，也就是同一个ThreadLocal，则直接更新该位置的数据。如果key不同再使用线性探测法。

如果在探测过程中有遇到key为null的数据则会进行一轮探测式清理的过程。

探测式清理：

从当前key为null的位置开始向前迭代查找其他过期的数据(key == null)，此时slotToExpunge和staleSlot的值是当前key为null的位置，直到遇到一个还没有放数据的位置结束，更新slotToExpunge到该位置。

接着再从最初位置向后迭代，此时新的数据由于哈希冲突还没有加入到里面。向后找是否有相同key(同一个ThreadLocal)的数据。如果找到了则更新这个位置的数据，并且 将staleSlot更新到这个位置。

最后开始进行清理工作，从slotToExpunge开始向后查找过期数据并清理。

如果向后查找的过程中没有遇到相同key的数据，则创建新的Entry替换staleSlot位置的数据。

ThreadLocal的扩容

如果上面的清理工作没有清理任何数据且Entry数量已经达到总长度的2/3，则开始rehash()操作；
这时再进行一次探测清理工作，这时再判断数量是否已经达到总长度的3/4；

扩容为原来的2倍，然后重新计算hash位置，放到新的扩容数组中。

ThreadLocal内存泄漏的问题

ThreadLocalMap的key是弱引用，value是强引用。所以在垃圾回收的时候会出现key被清理而value还保留的情况。

强引用：垃圾回收不会回收的对象，比如new一个对象的引用。如果错误保持强引用会产生内存泄漏。
软引用：当内存不足时，回收软引用对象。
弱引用：进行垃圾回收就会回收弱引用对象。
虚引用：

线程池ThreadPoolExecutor

每创建一个线程就会占用一定的资源，如果在高并发的情况下突然需要创建很多线程，则对内存的需求很大，有可能会出现OutOfMemoary。
而且线程并不是越多越好，因为CPU核心数只有那么多，线程太多上下文切换消耗大。

线程池状态：
ThreadPoolExecutor使用了int的高三位来表示线程池状态，低29位表示线程数量

状态名	高3位	接收新任务	处理阻塞队列任务	说明
RUNNING	111	Y	Y	
SHUTDOWN	000	N	Y	不会接收新任务，但会处理阻塞队列中的任务
STOP	001	N	N	会中断正在执行的任务，并抛弃阻塞队列中的任务
TIDYING	010	-	-	任务全部执行完毕，活动线程为0，即将进入终结
TERMINATED	011	-	-	终结状态

为什么不使用两个整数来分别表示状态和线程个数？

因为这些信息都是需要共享的，所以应该存放在原子变量中，但是如果使用两个原子变量就需要进行两次cas操作。如果能将状态和线程数量合并，则就只需要1次cas即可进行修改。

ThreadPoolExecutor构造参数

```
public ThreadPoolExecutor(int corePoolSize,
                          int maximumPoolSize,
                          long keepAliveTime,
                          TimeUnit unit,
                          BlockingQueue<Runnable> workQueue,
                          ThreadFactory threadFactory,
                          RejectedExecutionHandler handler)
//corePoolSize：核心线程数目
//maximumPoolSize：最大线程数目(核心线程数+救急线程数)
//keepAliveTime：生存时间(针对救急线程)
//unit：时间单位(针对救急线程)
//workQueue：阻塞队列
//threadFactory：线程工厂(可以为线程起名)
//handler：拒绝策略
```

救急线程：

一个新的任务t来了，如果此时核心线程都在使用且阻塞队列已满，这时还不会触发拒绝策略，而是先看有没有救急线程可以拿来使用，如果有则把t交给救急线程，否则触发拒绝策略。

keepAliveTime和unit都是对救急线程存活时间的控制：

救急线程在任务执行完毕后，会等待keepAliveTime的时间。超出keepAliveTime时间后会结束线程，节省资源。只有等到下次再有同样的情况出现的时候才会去重新创建救急线程。

ThreadPoolExecutor拒绝策略

```
AbortPolicy：让调用者抛出RejectWxecutionException异常。   这个是默认策略
CallerRunsPolicy：让调用者自己去执行任务
DiscardPolicy：让调用者放弃本次任务
DiscardOldestPolicy：放弃队列中最早的任务，由当前任务取代

框架提供的实现：
Dubbo：在抛出RejectWxecutionException异常之前会记录到日志中，并dump线程栈信息，方便定位问题
Netty：创建了一个新的线程执行任务
ActiveMQ：带超时等待(60s)的放入队列
PinPoint：使用一个拒绝策略链，会逐一尝试策略链中的每种策略
```

各种线程池

```
/*
    newFixedThreadPool
        核心线程数==最大线程数(没有救急线程)，也无需设置存活时间
        阻塞队列没有设置长度，表示可以存放任意数量的任务
    适用于：任务量已知，相对耗时的任务
*/
public static ExecutorService newFixedThreadPool(int nThreads) {
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}

/*
    newCachedThreadPool
        核心线程数是0，最大线程数是Integer.MAX_VALUE，救急线程的存活时间是60s。这表明全部都是救急线程(60s后被回收)，救急线程可以无限创建
```

```
        队列采用SynchronousQueue，将任务放入队列中，只有在该任务被取走的时候，放入方法才算结束。且没有设置容量
    适用于：任务数计较密集，但每个任务执行时间短的情况
*/
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}

/*
    newSingleThreadExecutor
        希望多个任务排队执行，线程数固定为1，当任务多于1的时候会放入容量无限的队列中等待。
    和自己创建一个线程执行任务的区别：
        自己创建的线程如果遇到异常会结束。但是在单线程线程池中即使遇到异常，后面的任务还是会创建另外一个线程去继续执行。
*/
public static ExecutorService newSingleThreadExecutor() {
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

线程池提交方法

```
// 执行任务，Runnabl没有返回结果，无法判断任务是否被线程执行成功
void execute(Runnable command);

// 提交任务 task，用返回值 Future 获得任务执行结果，可以通过Future判断任务是否执行成功
<T> Future<T> submit(Callable<T> task);

// 提交 tasks 中所有任务
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
        throws InterruptedException;

// 提交 tasks 中所有任务，带超时时间
<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks,
                              long timeout, TimeUnit unit)
        throws InterruptedException;

// 提交 tasks 中所有任务，哪个任务先成功执行完毕，返回此任务执行结果，其它任务取消
<T> T invokeAny(Collection<? extends Callable<T>> tasks)
        throws InterruptedException, ExecutionException;

// 提交 tasks 中所有任务，哪个任务先成功执行完毕，返回此任务执行结果，其它任务取消，带超时时间
<T> T invokeAny(Collection<? extends Callable<T>> tasks,
                    long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
```

关闭线程池

```
shutdown():
    线程池状态变为SHUTDOWN
        不会再接收新任务；
        已提交的任务会执行完；
        不会阻塞线程的执行；

shutdownNow():
    线程池状态变为STOP
        不会接收新任务；
        将阻塞队列中的任务返回；
        使用interrupt中断正在执行的任务
```

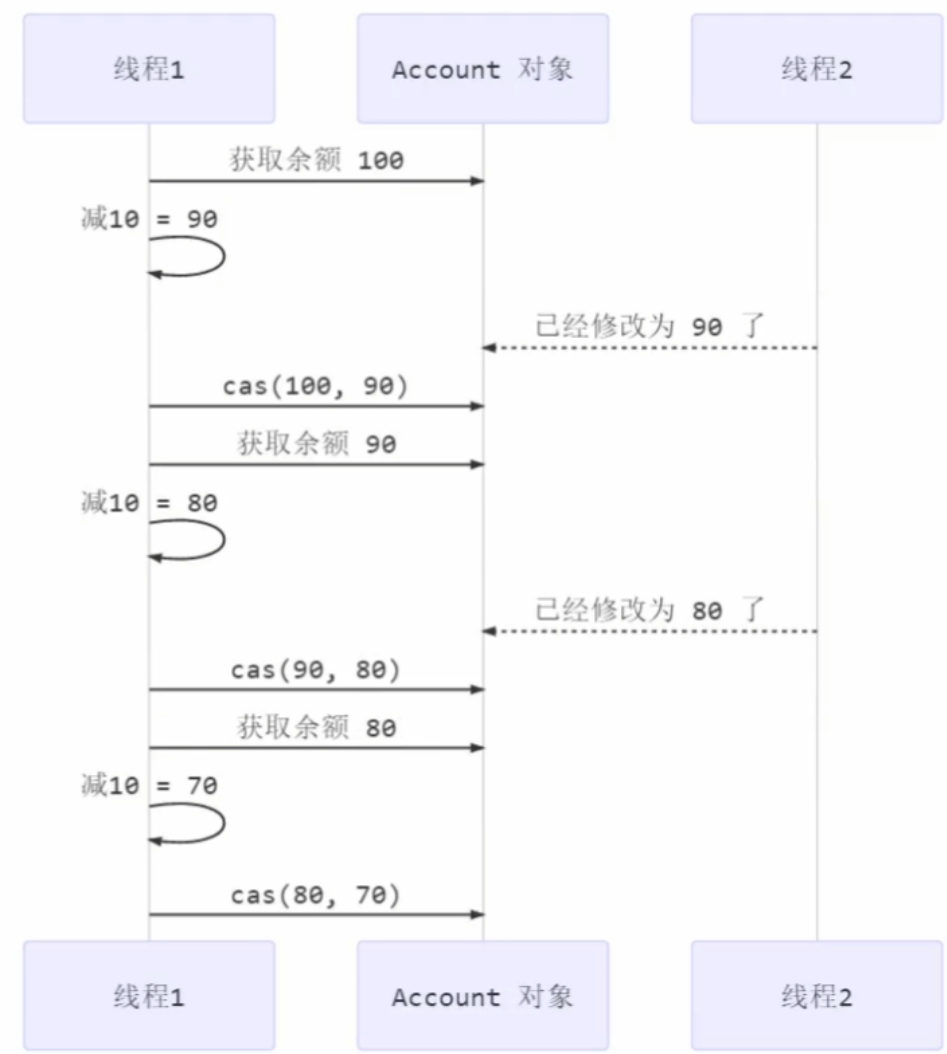
tomcat线程池

```
LimitLatch：用来限流，可以控制tomcat的最大连接个数
Acceptor：负责接收新的socket连接
Poller：监听socket channel是否有可读性I/O事件。如果有，则封装一个任务对象交给Executor线程池处理
Executor：处理请求

tomcat分为两个部分：Connector和Executor
Connector配置：
    配置项                      默认值              说明
    acceptorThreadCount         1                   acceptor线程数量
    pollerThreadCount           1                   poller线程数量
    minSpareThreads             10                  核心线程数(corePoolSize)
    maxThreads                  200                 最大线程数(maximumPoolSize)
    executor                    -                   Executor名称设置
Executor配置
    配置项                          默认值                  说明
    threadPriority                  5                       线程优先级
    daemon                          true                    是否守护线程
    minSpareThreads                 25                      核心线程数(corePoolSize)
    maxThreads                      200                     最大线程数
(maximumPoolSize)
    maxIdleTime                     60000                   线程生成时间，单位是毫秒，默认1分钟
    maxQueueSize                    Integer.MAX_VALUE       队列长度
    prestartminSpareThreads         false                   核心线程是否在服务器启动时启动
```

Fork/Join线程池

```
jdk1.7加入的新的线程池实现，体现一种分治的思想，适用于能进行任务拆分的cpu密集型运算

Fork/Join在分治的基础上加入了多线程，可以把每个任务的分解和合并交给不同的线程来完成，进一步提升运算效率

Fork/Join会默认创建与cpu核心数大小相同的线程池
```

CAS比较并交换



底层需要volatile的支持，因此是要获取最新的值来比较的，所以加入volatile保证可见性。

无锁比加锁的效率高主要体现在线程上下文切换。

特点：

结合CAS和volatile可以实现无锁并发，适用于线程数少、多核CPU的场景下。
CAS是基于乐观锁的思想：只有其他线程没有修改数据，才能修改；如果其他线程修改了数据就重试。
synchronized是基于悲观锁的思想：只有上锁的线程才能读写数据。
CAS是无锁并发、无阻塞并发：
    因为没有使用synchronized，所以线程不会进入阻塞，减少线程上下文切换；
    但是如果竞争很激烈(线程数过多)，会导致重试频繁发生，效率也会受影响。

Atomic原子类

常说的JUC包：java.util.concurrent 这是java处理并发的包

JUC包中的4类原子类：
基本类型：AtomicInteger、AtomicLong、AtomicBoolean
数组类型：AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray
引用类型：AtomicReference、AtomicStampedReference、AtomicMarkableReference
对象的属性修改类型：AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater

使用Atomic原子类，即使不加锁也能实现线程安全

一般的原子类型有ABA问题，可使用AtomicStampedReference解决。只要有其他线程改动过了共享变量，自己线程的cas就失败。这里增加了版本号，可知道更改次数。
使用AtomicMarkableReference，只要被其他线程更改过，就cas失败。只关心是否更改过。

AtomicInteger的原理

主要使用了CAS+volatile　　Unsafe类

```
class AccountCas implements Account{
    private AtomicInteger balance;

    public AccountCas(int balance){
        this.balance = new AtomicInteger(balance);
    }

    @Override
    public Integer getBalance(){
        return balance.get();
    }

    @Override
    public void withdraw(Integer amount){
        //方式1
        //while(true){
        //    int prev = balance.get();
        //    int next = prev - amount;
        //    if(balance.compareAndSet(prev, next)){
        //        break;
        //    }

        //方式2
        balance.getAndAdd(-1 * amount);
        }
    }
}
```

LongAdder原子累加器

一般的原子类型做CAS操作都是通过while(true)来判断值是否被修改，因此效率不太高。LongAdder在底层增加了单元数组，让多个线程对不同的数组单元进行累加，这样减少cas次数，提升效率

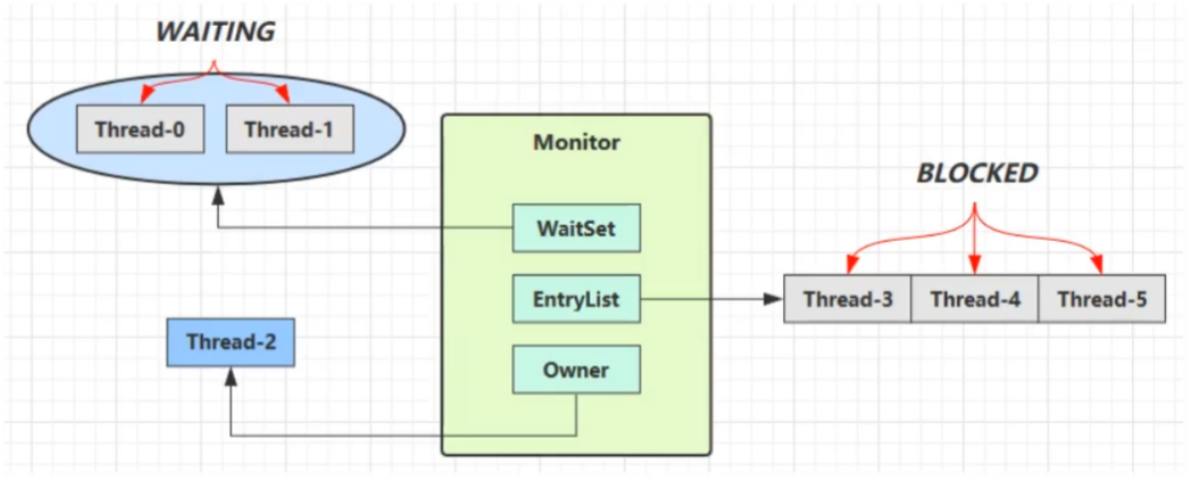
Unsafe

Unsafe对象提供了非常底层的操作内存、线程的方法。Unsafe无法直接调用，需要使用反射。

sleep和wait

1. sleep是Thread方法，wait是Object方法；
2. sleep不需要强制和synchronized配合使用，wait需要和synchronized一起使用；
3. sleep在睡眠的时候不会释放锁，wait在等待的时候会释放对象锁。

wait/notify



占同Monitor的线程发现执行条件并不满足，为了不占用资源，则调用wait()方法，并且该线程进入WaitSet中，线程变为WAITING状态。

注意：

- BLOCKED和WAITING都属于阻塞状态，不会占用CPU的时间片；
- BLOCKED线程会在Owner线程释放锁的时候被唤醒；
- WAITING线程会在Owner线程调用notify或者notifyAll的时候被唤醒，但唤醒后还是要进入EntryList中竞争；
- 如果有多个线程在等待唤醒，notify唤醒的是随机的一个；使用notifyAll可以叫醒全部等待的线程.

```
//线程1
synchronized(lock){
    while(条件不成立){
        lock.wait();
    }
    //条件成立，继续做事情
}

//线程2
synchronized(lock){
    lock.notifyAll();
}
```

park/unpark原理

每一个线程都有自己的一个Parker对象，该对象有三部分组成：counter、cond、mutex.

如果线程t1直接调用LockSupport.park():

先检查counter是否为0，如果不为0则继续执行线程任务，不阻塞；

counter为0后，获得mutex锁；

线程进入cond中阻塞；

设置counter为0.

接下来t1调用LockSupport.unpark():

设置counter为1；

唤醒cond中的阻塞线程；

t1恢复运行；

设置counter为0；

如果一开始直接就调用LockSupport.unpark():

仅设置counter为1；

然后调用LockSupport.park():

检查counter，因为不为0，则继续运行，没有阻塞；

设置counter为0.

park/unpark对比wait/notify

1. wait、notify、notifyAll必须配合Monitor一起使用，也就是synchronized，而park、unpark不用
2. park、unpark在LockSupport类中，wait、notify、notifyAll在Object类中
3. park、unpark以线程为单位来唤醒和阻塞线程，而notify只能随机唤醒一个线程
4. park、unpark可以先unpark，而wait、notify不能先notify

```
//阻塞线程
LockSupport.park(t1);

//唤醒线程
LockSupport.unpark(t1);
```

线程状态转换

1. NEW ----> RUNNABLE

   调用`t.start()`方法后，变为RUNNABLE

2. RUNNABLE ----> WAITING

```
t线程调用synchronized(obj)获取了对象锁后
    调用obj.wait()方法，t线程从RUNNABLE ----> WAITING
        调用obj.notify()、obj.notifyAll()、t.interrupt()方法时：
            竞争成功，t线程从WAITING ----> RUNNABLE
            竞争失败，t线程从WAITING ----> BLOCKED

如果在主线程中调用t.join()方法时，则主线程从RUNNABLE ----> WAITING，主线程会在t线程对象的监视器上等待
    t线程运行结束或者主线程调用interrupt方法，主线程从WAITING ----> RUNNABLE

当前线程调用LockSupport.park()方法会让当前线程从RUNNABLE ----> WAITING
    主线程调用LockSupport.unpark(目标线程)方法，或者目标线程调用interrupt方法，会让目标线程从WAITING ----> RUNNABLE
```

3. RUNNABLE ----> TIMED_WAITING

```
t线程调用synchronized(obj)获取了对象锁后
    调用obj.wait(long n)方法时，t线程从RUNNABLE ----> TIMED_WAITING
        t线程等待超过n毫秒，或者调用obj.notify()、obj.notifyAll()、t.interrupt()方法时：
            竞争成功，t线程从TIMED_WAITING ----> RUNNABLE
            竞争失败，t线程从TIMED_WAITING ----> BLOCKED

如果在主线程中调用t.join(long n)方法时，则主线程从RUNNABLE ----> TIMED_WAITING，主线程会在t线程对象的监视器上等待
    主线程等待时间超过了n毫秒或者t线程运行结束或者主线程调用interrupt方法，主线程从TIMED_WAITING ----> RUNNABLE

当前线程调用LockSupport.parkNanos(long nanos)或者调用LockSupport.parkUntil(long millis)会让当前线程从RUNNABLE ----> TIMED_WAITING
    主线程调用LockSupport.unpark(目标线程)方法，或者目标线程调用interrupt方法或者目标线程等待超时，会让目标线程从TIMED_WAITING ----> RUNNABLE

当前线程调用Thread.sleep(long n)，当前线程从RUNNABLE ----> TIMED_WAITING
    当前线程等待超过n毫秒，当前线程从TIMED_WAITING ----> RUNNABLE
```

4. RUNNABLE ----> BLOCKED

```
t线程使用synchronized(obj)获取锁对象时，如果竞争失败，则会从RUNNABLE ----> BLOCKED
    当获得obj锁的线程执行完毕后，会唤醒该对象上所有BLOCKED的线程，重新竞争。如果t线程竞争成功，则从BLOCKED ----> RUNNABLE,其他失败的仍然是BLOCKEWD
```

5. RUNNABLE ----> TERMINATED

interrupt

打断sleep、wait、join的线程，这些线程都是阻塞状态

```java
public static void test() throws InterruptedException {
    Thread t1 = new Thread(() -> {
        log.debug("sleep.........");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
    }, "t1");

    t1.start();
    Thread.sleep(1000);
    log.debug("interrupt........");
    t1.interrupt();
    log.debug("打断标记：{}", t1.isInterrupted()); //阻塞的线程打断后标记为false，且抛出异常
}
```

打断正常运行的线程(并没有真正打断，而是将打断标记置为true，打断还是需要线程自己内部进行停止)

```
public static void test1() throws InterruptedException {
    Thread t1 = new Thread(() -> {
        while (true){
            if (Thread.currentThread().isInterrupted()){
                break;
            }
        }
    }, "t1");
    t1.start();
    Thread.sleep(1000);
    log.debug("start interrupt...");
    t1.interrupt();         //interrupt只是告诉线程需要打断了，但是并没有真的打断，需要线程内部通过打断标记自行中断
    log.debug("end interrupt...");
}
```

打断park线程：不会清空打断状态

```
private static void test3() throws InterruptedException {
    Thread t1 = new Thread(() -> {
        log.debug("park...");
        LockSupport.park();
        log.debug("unpark...");
        log.debug("打断状态：{}", Thread.currentThread().isInterrupted());   //输出true
    }, "t1");
    t1.start();
    sleep(0.5);
    t1.interrupt();
}
```

如果打断标记已经是true，则park会失效

```
private static void test4() {
    Thread t1 = new Thread(() -> {
        for (int i = 0; i < 5; i++) {
            log.debug("park...");
            LockSupport.park();
            log.debug("打断状态：{}", Thread.currentThread().isInterrupted());
        }
    });
    t1.start();
    sleep(1);
```

```
    t1.interrupt();
}

//输出
21:13:48.783 [Thread-0] c.TestInterrupt - park...
21:13:49.809 [Thread-0] c.TestInterrupt - 打断状态：true
21:13:49.812 [Thread-0] c.TestInterrupt - park...
21:13:49.813 [Thread-0] c.TestInterrupt - 打断状态：true
21:13:49.813 [Thread-0] c.TestInterrupt - park...
21:13:49.813 [Thread-0] c.TestInterrupt - 打断状态：true
21:13:49.813 [Thread-0] c.TestInterrupt - park...
21:13:49.813 [Thread-0] c.TestInterrupt - 打断状态：true
21:13:49.813 [Thread-0] c.TestInterrupt - park...
21:13:49.813 [Thread-0] c.TestInterrupt - 打断状态：true
```

join

如果有一个线程需要等待另外一个线程执行完毕后再执行，则可以使用join方法。

join底层等价于：

```
synchronized(t1){
    while(t1.isAlive()){
        t1.wait(0);
    }
}
```

java内存模型JMM

1. 原子性：保证指令不会受到线程上下文切换的影响
2. 可见性：保证指令不会受到cpu缓存的影响

```
//volatile:每次都到主存中获取值，避免线程从自己的私有缓存中获取，但是volatile无法保证原子性。一般用在一个写线程，多个读线程的情况
public static void test() throws InterruptedException {
    new Thread(() -> {
        log.debug("t1线程开始工作");
        while (run){

        }
    }, "t1").start();

    Thread.sleep(1000);
    log.debug("t1线程结束...");
    run = false;
}

//可见性问题也可以使用synchronized解决，synchronized也保证原子性，但是性能比volatile低
public static void test1() throws InterruptedException {
    new Thread(() -> {
        log.debug("t1线程开始工作");
        while (true){
            synchronized (lock){
                if (!run){
                    break;
                }
```

```
            }
        }
    }, "t1").start();

    Thread.sleep(1000);
    log.debug("t1线程结束...");
    run = false;
}
```

3. 有序性：保证指令不会受cpu指令并行优化的影响

```
//JVM在不影响正确性的前提下，可以调整语句的执行顺序
static int i;
static int j;

i = 1;
j = 2;

//这里的执行顺序可以的先i后j，也可以是先j后i，这也被称为指令重排。能提升效率
//但是在多线程下，指令重排就会影响正确性。可以使用volatile来禁止指令重排
```