# 一、 synchronized

Jdk1.6之后进行了优化，无锁——》偏向锁——》轻量级锁——》重量级锁

Synchronized与lock的区别？

1、用法不一样.在需要同步的对象中加入synchronized控制,synchronized既可以加载方法上,也可以加在特定代码块中,括号中表示需要锁的对象.而Lock需要显式的指定起始位和终止位置.synchronized是托管给JVM执行的,而Lock的锁定是通过代码实现的,它有比synchronized更精确的线程语义.

2)性能不一样.在JDK5中增加了一个Lock接口实现类ReentrantLock.它不仅拥有和synchronized相同的并发性和内存语义,还多了锁投票,定时锁,等候和中断锁等.它们的性能在不同的情况下会有不同:在资源竞争不是很激烈的情况下,synchronized的性能要优于ReentrantLock,但是在资源竞争很激烈的情况下,synchronized的性能会下降得非常快,而ReentrantLock的性能基本保持不变.

3)锁机制不一样,synchronized获得锁和释放的方式都是在块结构中,当获取多个锁时,必须以相反的顺序释放,并且是自动解锁,不会因为出来异常而导致锁没有被释放从而引发死锁.而Lock则需要开发人员手动去释放,并且必须在finally块中释放,否则会引起死锁问题的发生.此外,Lock还提供了更强大的功能,它的tryLock()方法可以采用非阻塞的方式去获取锁.

**高并发场景下lock性能高的原因？**

高并发场景下synchronized升级为重量级锁，不可逆转，新来的线程全部被阻塞，而使用lock在阻塞之前仍然会自旋尝试获取锁，失败后才会进行阻塞

Synchronized原始采用的是CPU 悲观锁机制，即线程获得的是独占锁，独占锁意味着其它线程只能依靠阻塞来等待线程释放锁，而CPU转换线程阻塞时会引起上下文切换（用户态与内核态的切换），当有很多线程竞争锁时，频繁的上下文切换使效率不高，对比lock，synchronized唤醒线程时会唤醒所有阻塞线程，而lock只会唤醒同步队列的头结点一个线程

Lock使用的乐观锁模式，即每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止，通过cas实现乐观锁机制

## 1.1 synchronized的应用

1、方法

静态方法（相当于加锁加在类对象上）、非静态方法（加锁加在当前this类对象上），此时如果类对象由spring容器管理则必须设置为单例

1. 代码块

对象锁、类锁

3、当作用的对象为静态时，则作用于整个类

4、跨方法加对象锁可以使用unsafe的moniterEnter与monitorExit来实现

5、synchronized从八大内存操作的Lock开始，volatile则从read开始

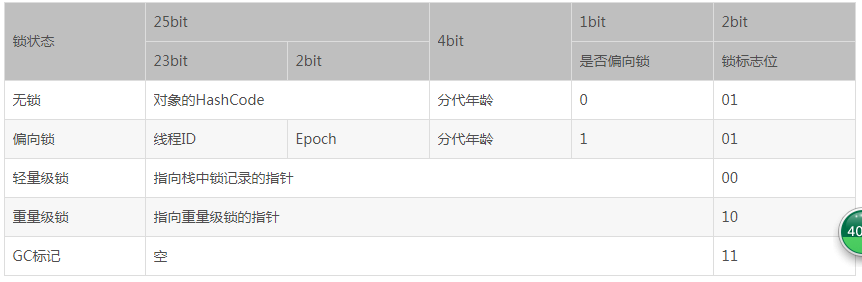
## 1.2对象的组成，与对象头详解

**1、对象组成**

Java对象由三部分组成，对象头（Header）、实例数据（Instance Data）、对齐填充字节（Padding）

**A、对象头**

对象头由Mark Word（32位系统32位，64位系统64位）、指向类的指针、数组长度（只有数组对象有）



**垃圾回收age最大为15，因为存储分代年龄的空间4位，最大值为15**

#### 2、指向类的的指针（class pointer）

这一部分用于存储对象的类型指针，该指针指向它的类元数据，JVM通过这个指针确定对象是哪个类的实例。该指针的位长度为JVM的一个字大小，即32位的JVM为32位，64位的JVM为64位。  
如果应用的对象过多，使用64位的指针将浪费大量内存，统计而言，64位的JVM将会比32位的JVM多耗费50%的内存。为了节约内存可以使用选项+UseCompressedOops开启指针压缩，其中，oop即ordinary object pointer普通对象指针。开启该选项后，下列指针将压缩至32位：

1. 每个Class的属性指针（即静态变量）
2. 每个对象的属性指针（即对象变量）
3. 普通对象数组的每个元素指针

当然，也不是所有的指针都会压缩，一些特殊类型的指针JVM不会优化，比如指向PermGen的Class对象指针(JDK8中指向元空间的Class对象指针)、本地变量、堆栈元素、入参、返回值和NULL指针等。

1. 数组长度

如果对象是一个数组，那么对象头还需要有额外的空间用于存储数组的长度，这部分数据的长度也随着JVM架构的不同而不同：32位的JVM上，长度为32位；64位JVM则为64位。64位JVM如果开启+UseCompressedOops选项，该区域长度也将由64位压缩至32位。

1. **实例数据**

真正存储有效信息，即程序代码中所定义的各种类型的字段内容，包括父类继承下来的和子类定义的。这部分存储顺序受虚拟机分配策略参数（FiledsAllocationStyle)和字段在Java源码中定义的顺序的影响。HotSpot虚拟机默认分配策略为longs/double、ints、shorts/chars、bytes/booleans、oops(Ordinary Object Pointers)，从分配策略可以看出，相同宽度的字段总是被分配到一起。在此前提条件下，父类中定义的变量会出在子类之前，若CompactFileds的参数值为true，那么子类中较**窄**的变量也可能会插到父类变量的空隙中。

1. **填充字节**

不是必然存在的，仅起着占位符作用。HotSpot虚拟机的自动内存管理系统要求对象起始地址必须是8字节的整数倍，即对象大小必须是8字节的整数倍，而对象头正好是8字节的整数倍，而对象实例数据部分没有对齐，需要通过对齐填充来补全。

## 1.3synchronized的实现

在Hot Spot中，monitor是由ObjectMonitor类来实现的

其中有两个队列 \_EntryList和 \_WaitSet，它们是用来保存ObjectMonitor对象列表， \_owner指向持有ObjectMonitor对象的线程。 当多个线程访问同步代码时，线程会进入\_EntryList区，当线程获取对象的monitor后(对于线程获得锁的优先级，还有待考究)进入 \_Owner区并且将 \_owner指向获得锁的线程(monitor对象被线程持有)， \_count++，其他线程则继续在 \_EntryList区等待。若线程调用wait方法，则该线程进入 \_WaitSet区等待被唤醒。线程执行完后释放monitor锁并且对ObjectMonitor中的值进行复位。 上面说到synchronized使用的锁都放在对象头里，大概指的就是Mark Word中指向互斥量的指针指向的monitor对象内存地址了。

## 1.4各个锁概念及运行流程

**JVM一般是这样使用锁和Mark Word的：**

1，当没有被当成锁时，这就是一个普通的对象，Mark Word记录对象的HashCode，锁标志位是01，是否偏向锁那一位是0。

2，当对象被当做同步锁并有一个线程A抢到了锁时，锁标志位还是01，但是否偏向锁那一位改成1，前23bit记录抢到锁的线程id，表示进入偏向锁状态。

3，当线程A再次试图来获得锁时，JVM发现同步锁对象的标志位是01，是否偏向锁是1，也就是偏向状态，Mark Word中记录的线程id就是线程A自己的id，表示线程A已经获得了这个偏向锁，可以执行同步锁的代码。

4，当线程B试图获得这个锁时，JVM发现同步锁处于偏向状态，但是Mark Word中的线程id记录的不是B，那么线程B会先用CAS操作试图获得锁，这里的获得锁操作是有可能成功的，因为线程A一般不会自动释放偏向锁。如果抢锁成功，就把Mark Word里的线程id改为线程B的id，代表线程B获得了这个偏向锁，可以执行同步锁代码。如果抢锁失败，则继续执行步骤5。

5，偏向锁状态抢锁失败，代表当前锁有一定的竞争，偏向锁将升级为轻量级锁。JVM会在当前线程的线程栈中开辟一块单独的空间，里面保存指向对象锁Mark Word的指针，同时在对象锁Mark Word中保存指向这片空间的指针。上述两个保存操作都是CAS操作，如果保存成功，代表线程抢到了同步锁，就把Mark Word中的锁标志位改成00，可以执行同步锁代码。如果保存失败，表示抢锁失败，竞争太激烈，继续执行步骤6。

6，轻量级锁抢锁失败，JVM会使用自旋锁，自旋锁不是一个锁状态，只是代表不断的重试，尝试抢锁。从JDK1.7开始，自旋锁默认启用，自旋次数由JVM决定。如果抢锁成功则执行同步锁代码，如果失败则继续执行步骤7。

7，自旋锁重试之后如果抢锁依然失败，同步锁会升级至重量级锁，锁标志位改为10。在这个状态下，未抢到锁的线程都会被阻塞。

**偏向锁的获取与释放**

1、访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01，确认为可偏向状态。

2、如果为可偏向状态，则测试线程ID是否指向当前线程，如果是，进入步骤5，否则进入步骤3。

3、如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行5；如果竞争失败，执行4。

4、如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点（safepoint，无字节码被执行）时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。（撤销偏向锁的时候会导致stop the word）

5、执行同步代码。

**1、轻量级锁的加锁过程**

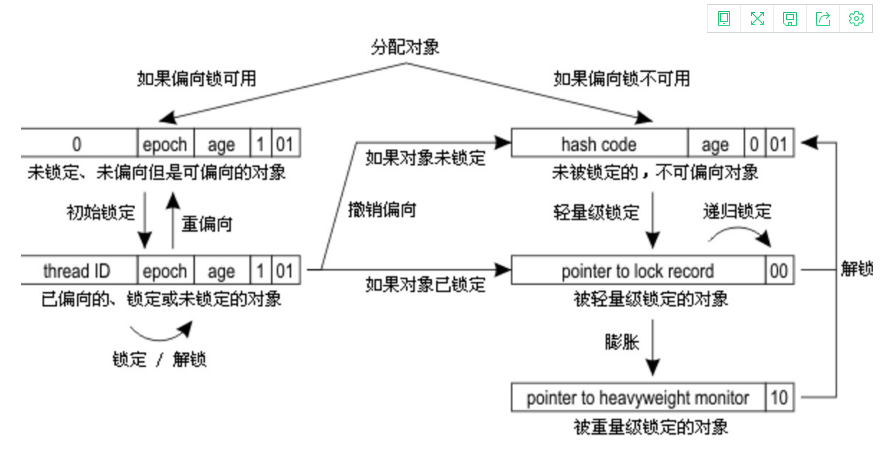
　　（1）在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为 Displaced Mark Word。这时候线程堆栈与对象头的状态如图2.1所示。

　　（2）拷贝对象头中的Mark Word复制到锁记录中。

　　（3）拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向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）。

　　（4）如果这个更新动作成功了，那么这个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，这时候线程堆栈与对象头的状态如图2.2所示。

（5）如果这个更新操作失败了，虚拟机首先会检查对象的Mark 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，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，后面等待锁的线程也要进入阻塞状态。 而当前线程便尝试使用自旋来获取锁，自旋就是为了不让线程阻塞，而采用循环去获取锁的过程。



## 1.5 synchronized原理

Synchronized底层依赖8大原子操作的lock与unlock指令实现

Synchronized是可重入锁，owern置空后其它线程即可进入

Synchronized升级为重量级锁时需要申请互斥量，需要调用内核，未获得锁的所有线程由用户态转为内核态，被挂起

**锁的粗化**：当有多个连续的加锁操作时，jvm将多个加锁操作合并为一个，增大锁的粒度

**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。

**逃逸分析：**

一、 同步省略。如果一个对象被发现只能从一个线程被访问到，那么对于这个对象的操作可以不考虑同步。  
二、 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永远不会逃逸，对象可能是栈分配的候选，而不是堆分配。  
三、 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，而是存储在CPU寄存器中。

**Java实例对象存储在哪？**

1、如果实例对象存储在堆区时：实例对象内存存在堆区，实例的引用存在栈上，实例的元数据class存在方法区或者元空间；

2、如果无逃逸行为则对象可能存放于栈空间；

3、对象部分或全部存放于寄存器中。

**标量替换**：

标量替换，scalar replacement。Java中的原始类型无法再分解，可以看作标量（scalar）；指向对象的引用也是标量；而对象本身则是聚合量（aggregate），可以包含任意个数的标量。如果把一个Java对象拆散，将其成员变量恢复为分散的变量，这就叫做标量替换。拆散后的变量便可以被单独分析与优化，可以各自分别在活动记录（栈帧或寄存器）上分配空间；原本的对象就无需整体分配空间了

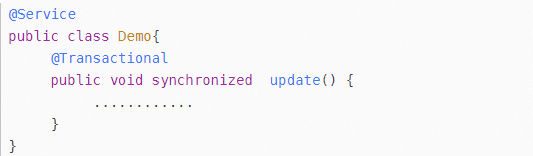
标量即不可被进一步分解的量，而JAVA的基本数据类型就是标量（如：int，long等基本数据类型以及reference类型等），标量的对立就是可以被进一步分解的量，而这种量称之为聚合量。而在JAVA中对象就是可以被进一步分解的聚合量

通过逃逸分析确定该对象不会被外部访问，并且对象可以被进一步分解时，JVM不会创建该对象，而会将该对象成员变量分解若干个被这个方法使用的成员变量所代替。这些代替的成员变量在栈帧或寄存器上分配空间。

我们通过JVM内存分配可以知道JAVA中的对象都是在堆上进行分配，当对象没有被引用的时候，需要依靠GC进行回收内存，如果对象数量较多的时候，会给GC带来较大压力，也间接影响了应用的性能。为了减少临时对象在堆内分配的数量，JVM通过逃逸分析确定该对象不会被外部访问。那就通过标量替换将该对象分解在栈上分配内存，这样该对象所占用的内存空间就可以随栈帧出栈而销毁，就减轻了垃圾回收的压力

## 1.6 synchronized面试题

1、synchronized与transactional同时使用失效的问题



此方法在中先开启事物，之后再加锁，代码执行完之后释放锁，然后提交事物，在高并发场景下，其他线程在上一个线程事物提交之前又去数据库查询数据，之后上一线程修改的数据的事物物才被提交，从而造成数据被重复消费的情况，解决办法在事物之上加锁即提高锁的粒度，transactional事物的提交是同步的不用考虑这一部分

# 二、读写锁 ReadWriteLock

读写锁实现读读不影响，读写、写写互斥，ReadWriteLock管理一组锁，一个只是读锁，一个是写锁，java并发库中ReetrantReadWriteLock实现ReadWriteLock并加入可重入特性，

**ReetrantReadWriteLock的特性：**

1. **获取锁的顺序**
2. 非公平模式（默认）

当以非公平初始化时，读锁和写锁的获取的顺序是不确定的。非公平锁主张竞争获取，可能会延缓一个或多个读或写线程，但是会比公平锁有更高的吞吐量。

（原理，公平模式会维护一个队列，所有线程会按照先后顺序依次执行，而非公平模式则没有执行顺序，后面的线程可能先获取锁，从而省去线程解除休眠状态的时间）

**B、公平模式**

当以公平模式初始化时，线程将会以队列的顺序获取锁。当当前线程释放锁后，等待时间最长的写锁线程就会被分配写锁；或者有一组读线程组等待时间比写线程长，那么这组读线程组将会被分配读锁。

1. **可重入**

可重入锁指一个线程获取某个锁后，还可以继续获取该锁，即允许线程多次获取同一个锁，synchronized内置锁就是可以重入的，如果A类有两个synchronized方法，1、2，那么1调用2是允许的

package test;

import java.util.concurrent.locks.ReentrantReadWriteLock;

public class Test1 {

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

final ReentrantReadWriteLock lock = new ReentrantReadWriteLock ();

Thread t = new Thread(new Runnable() {

@Override

public void run() {

lock.writeLock().lock();

System.out.println("Thread real execute");

lock.writeLock().unlock();

}

});

lock.writeLock().lock();

lock.writeLock().lock();

t.start();

Thread.sleep(200);

System.out.println("realse one once");

lock.writeLock().unlock();

}

}

运行结果只有，realse one once，上面的代码会出现死锁，因为主线程2次获取了锁，但却只释放一次锁，导致线程永远不能获取锁，一个线程获取几次锁就必须释放几次锁，对于内置锁也适用，每一次进入和离开synchronization方法，就是一次完整的锁获取与释放

**读锁获取锁逻辑**

protected final int tryAcquireShared(int unused) {

/\*

\* Walkthrough:

\* 1. If write lock held by another thread, fail.

\* 2. Otherwise, this thread is eligible for

\* lock wrt state, so ask if it should block

\* because of queue policy. If not, try

\* to grant by CASing state and updating count.

\* Note that step does not check for reentrant

\* acquires, which is postponed to full version

\* to avoid having to check hold count in

\* the more typical non-reentrant case.

\* 3. If step 2 fails either because thread

\* apparently not eligible or CAS fails or count

\* saturated, chain to version with full retry loop.

\*/

Thread current = Thread.currentThread();

//获取锁状态，锁状态的高16为代表读锁的获取次数，低16位代表写锁的

int c = getState();

//获取读锁的获取锁次数，如果不为零，则看获取锁的线程与当前线程是否一致，不一致则失败

if (exclusiveCount(c) != 0 &&

getExclusiveOwnerThread() != current)

return -1;

int r = sharedCount(c);

//1、检查读请求是否要阻塞，2、读锁获取线程是否达到最大允许值3、使用cas修改锁状态

if (!readerShouldBlock() &&

r < MAX\_COUNT &&

compareAndSetState(c, c + SHARED\_UNIT)) {

if (r == 0) {// 标记是否为第一个加读锁的线程，并统计第一条线程重入次数

firstReader = current;

firstReaderHoldCount = 1;

} else if (firstReader == current) {

firstReaderHoldCount++;

} else {

ReentrantReadWriteLock.Sync.HoldCounter rh = cachedHoldCounter;

if (rh == null || rh.tid != getThreadId(current))

cachedHoldCounter = rh = readHolds.get();

else if (rh.count == 0)

readHolds.set(rh);

rh.count++;

}

return 1;

}

return fullTryAcquireShared(current);

}

**//写锁获取锁逻辑**

protected final boolean tryAcquire(int acquires) {

/\*

\* Walkthrough:

\* 1. If read count nonzero or write count nonzero

\* and owner is a different thread, fail.

\* 2. If count would saturate, fail. (This can only

\* happen if count is already nonzero.)

\* 3. Otherwise, this thread is eligible for lock if

\* it is either a reentrant acquire or

\* queue policy allows it. If so, update state

\* and set owner.

\*/

Thread current = Thread.currentThread();

int c = getState();

int w = exclusiveCount(c);

// c不等于零说明有线程获取了读锁或者写锁

if (c != 0) {

// (Note: if c != 0 and w == 0 then shared count != 0)

if (w == 0 || current != getExclusiveOwnerThread())

return false;

// 如果写锁获取量达到最大则抛异常

if (w + exclusiveCount(acquires) > MAX\_COUNT)

throw new Error("Maximum lock count exceeded");

// Reentrant acquire

setState(c + acquires);

return true;

}

//尝试获取锁

if (writerShouldBlock() ||

!compareAndSetState(c, c + acquires))

return false;

setExclusiveOwnerThread(current);

return true;

}

**3锁降级**

锁降级：从写锁变成读锁，锁升级：从读锁变为写锁

**ReetrantReadWriteLock**支持锁降级（由写锁直接转换为读锁），不支持升级（需要释放锁之后才能转换）

读写关系

package test;

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

/\*\*

\*

\* ReetrantReadWriteLock实现

\* @author itbird

\*

\*/public class ReadAndWriteLockTest {

public static ReentrantReadWriteLock lock = new ReentrantReadWriteLock();

public static void main(String[] args) {

//同时写

ExecutorService service = Executors.newCachedThreadPool();

service.execute(new Runnable() {

@Override

public void run() {

writeFile(Thread.currentThread());

}

});

service.execute(new Runnable() {

@Override

public void run() {

writeFile(Thread.currentThread());

}

});

}

// 读操作

public static void readFile(Thread thread) {

lock.readLock().lock();

boolean readLock = lock.isWriteLocked();

if (!readLock) {

System.out.println("当前为读锁！");

}

try {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(20);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(thread.getName() + ":正在进行读操作……");

}

System.out.println(thread.getName() + ":读操作完毕！");

} finally {

System.out.println("释放读锁！");

lock.readLock().unlock();

}

}

// 写操作

public static void writeFile(Thread thread) {

lock.writeLock().lock();

boolean writeLock = lock.isWriteLocked();

if (writeLock) {

System.out.println("当前为写锁！");

}

try {

for (int i = 0; i < 5; i++) {

try {

Thread.sleep(20);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(thread.getName() + ":正在进行写操作……");

}

System.out.println(thread.getName() + ":写操作完毕！");

} finally {

System.out.println("释放写锁！");

lock.writeLock().unlock();

}

}

}

\*\*\*\*\*\*\*boolean readLock = lock.isWriteLocked();判断当前锁类型

# 三、lock

reentrantLock的lock.lockInterruptibly()方法，提供了中断的功能，用于等待获取锁的线程结束等待状态

## 1、公平锁与非公平锁

ReentrantLock

tryAcquire(arg)🡪 hasQueuedPredecessors()

compareAndSetState(0, acquires)

公平锁-🡪acquire(1)🡪 { acquireQueued(addWaiter(Node.EXCLUSIVE), arg)

tryAcquire(arg)尝试获取锁资源，如果state为0，hasQueuedPredecessors()检测是否需要尝试获取锁return h != t && ((s = h.next) == null || s.thread != Thread.currentThread());当同步队列为空时尝试获取锁，当同步队列不为空，当前线程与节点对应的线程为同一线程时尝试获取锁，总结来说当同步队列非空，只有当前线程与头结点下一节点对应同一线程时才会尝试获取锁，以次保证线程获取锁的顺序性及公平锁。compareAndSetState(0, acquires)获取锁操作，如果获取锁成功则将独占线程设置为当前线程，如果state不为零，如果当前线程为独占线程则更新获取锁次数加1

acquireQueued(addWaiter(Node.EXCLUSIVE), arg)，当获取锁失败后会执行此方法，向同步队列添加独占类型的节点addWaiter(Node.EXCLUSIVE)，创建节点时将nextWaiter属性设置为空（此属性用于条件队列），通过cas添加节点compareAndSetTail(pred, node)，如果尾节点为空，则调用enq(node)对队列进行初始化，然后添加节点，acquireQueued 方法添加节点，node.predecessor()获取当前节点的前驱节点，如果为头结点则尝试获取锁，获取锁成功后将当前节点设置为头结点，将thread与prev属性置空，新增节点返回false，当前驱节点不是头结点，shouldParkAfterFailedAcquire判断节点的状态类型，signal状态返回true，如果大于零则将节点剔除掉node.prev = pred = pred.prev;，如果为其他状态则将其改为signal状态，在公平锁状态下只允许1与-1两种状态，通过自旋直至返回true，然后执行

parkAndCheckInterrupt()，通过unsafe的park方法阻塞当前线程（自己），然后自旋停止，获取锁的过程中出现异常则执行cancelAcquire(node)

公平锁的释放：release(int arg)-> tryRelease(int releases)

tryRelease(int releases)释放资源，首先将加锁次数减一，如果次数变为0，则将独占线程置空，如果不为0则更新加锁次数

unparkSuccessor(Node node)，判断头结点的后继结点是否正常等待的结点，如果是直接唤醒线程，如果不是则从尾部遍历整个链表找到离头结点最近的正常等待节点进行唤醒，没有剔除cancelled节点的动作

compareAndSetState(0, 1) 尝试获取锁资源，如果获取失败正则则行acquire方法

非公平锁：lock()—>{acquire（1）

acquire（1）方法 则执行tryAcquire(arg)尝试获取锁，调用nonfairTryAcquire(int acquires)直接重试获取锁或者再入锁，如果获取锁失败则调用

acquireQueued(addWaiter(Node.EXCLUSIVE), arg)新增节点，在此方法中通过lockSupport的park方法阻塞当前线程从而停止自旋

释放资源操作与公平锁一致

## 2、semaphore

semaphore.acquire()尝试获取所，tryAcquireShared(arg)直接获取锁，成功后执行逻辑，不成功即资源不足时则加入同步队列，执行doAcquireSharedInterruptibly(arg)，如果当前节点为头结点的后继节点则尝试获取锁，如果成功执行setHeadAndPropagate(node, r)，将当前结点设置为头结点，如果有资源剩余或者头结点为空或者等待状态为signal状态执行

doReleaseShared()唤醒线程，shouldParkAfterFailedAcquire(p, node) &&parkAndCheckInterrupt()进行线程的阻塞。

资源释放：semaphore.release()，releaseShared(int arg)，tryReleaseShared(arg)将资源加回资源池，doReleaseShared()具体的唤醒操作作，将头节点的状态改为0，然后改为PROPAGATE即-3，之所以不直接改为-3，因为unparkSuccessor(h)方法中会将节点状态改为0，所以直接改为-3的话后面要改为0然后改为-3，多了一步cas操作，unparkSuccessor(h)方法唤醒线程，唤醒的头结点的后继节点对应线程，线程唤醒后处于doAcquireSharedInterruptibly(arg)方法的自旋中，此时对应节点为头结点的后继节点，如果线程获取资源成功后，执行setHeadAndPropagate(node, r)的方法中继续唤醒后续线程，条件为资源仍有剩余，或者头结点为空或者头结点的状态为signal则继续唤醒后续线程，当然仍需满足头结点的后继节点为空或者后继节点为共享模式，，---总结，唤醒的逻辑仍然是顺序唤醒，前一节点获取资源成功后才继续唤醒后一节点，只有新进入的线程直接尝试获取资源的操作体现了非公平性。

## 3、ArrayBlockingQueue

创建阻塞队列时必须指定队列大小，且不能扩容，默认创建一个非公平锁，两个条件队列

this.items = new Object[capacity];

lock = new ReentrantLock(fair);

notEmpty = lock.newCondition();

notFull = lock.newCondition();

put()方法🡪1、首先获取重入锁lock.lockInterruptibly()，此方法线程被中断会抛异常

2、enqueue()数组未满时执行该方法，向数组新增元素，并通过notEmpty.signal()方法将消费者转移到同步队列，isHeldExclusively()该方法判断当前线程是否为独占线程，doSignal(Node first)将条件队列头节点移动到同步队列，循环执行直到条件队列为空，if ( (firstWaiter = first.nextWaiter) == null)，通过此判断条件将firstWaiter指向头结点的后继节点，以此来移除头结点，transferForSignal(first)具体移动头结点，if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))，尝试修改节点状态，如果不成功则代表该节点被cancelled，则继续转移下一节点，enq(node)将节点加入到同步队列的尾部，并返回该节点的前驱，如果前驱节点的的状态值大于零或者更改其状态值为signal失败，则唤醒当前节点

3、notFull.await()数组满了之后执行该方法，首先addConditionWaiter()将当前线程加入到条件队列，unlinkCancelledWaiters()清理被cancelled的节点，新元素插入到队尾，fullyRelease(node)一次性释放掉重入锁（多次），interruptMode = 0，正常节点未被中断，isOnSyncQueue(node)判断当前节点是否在同步队列中，如果不在则阻塞线程，checkInterruptWhileWaiting(node))更改interruptMode的值，是否被中断过，跳出while循环可能是加入到同步队列或者被中断，接下来尝试获取锁，获取锁成功后如果该节点在条件队列的后继节点不为空则清理队列，reportInterruptAfterWait(interruptMode)根据interruptMode的值做不同的处理

## 4、delayQueue

delayQueue的实现原理，其获取数据的方式有两种，take与poll，使用poll获取数据不会阻塞，没有数据直接返回null，使用take会阻塞，当队列为空时线程直接阻塞，当队列不为空时判断是否达到数据的延时时间，如果达到了直接返回，没有达到则判断leader线程属性是否为空，如果为空将当前线程设为leader线程，并将线程阻塞头节点到期所用时间，如果不为空则直接将线程阻塞，当leader线程触发获取到结点之后会唤醒一条线程继续拉取数据，当队列使用put或者offer添加数据时也会唤醒消费线程

## 5、读写锁

## 6、面试题

### 1、lock如何实现可见性功能的？

AQS中记录锁重入次数的变量state是由volatile修饰的变量，当对state修改之后，在进行unlock之前，会将工作做内存中的变量同步回主内存，从而实现可见性，依据的原理为happens-before 原则中的——如果线程1写入了volatile变量v,接着线程2读取了v，那么，线程1 写入v及之前的写操作都对线程2可见。也就是说，如果你感知到了volatile变量v的变化，那么在v之前的所有写操作你都可以感知的到。

# 无锁框架Disruptor

## 4.1 原理

Disruptor是一个开源框架，研发的初衷是为了解决高并发下列队锁的问题，最早由LMAX（一种新型零售金融交易平台）提出并使用，能够在无锁的情况下实现队列的并发操

作，并号称能够在一个线程里每秒处理6百万笔订单

**核心设计原理**

Disruptor通过以下设计来解决队列速度慢的问题：

环形数组结构：

为了避免垃圾回收，采用数组而非链表。同时，数组对处理器的缓存机制更加友好（回顾一下：CPU加载空间局部性原则）。

元素位置定位：

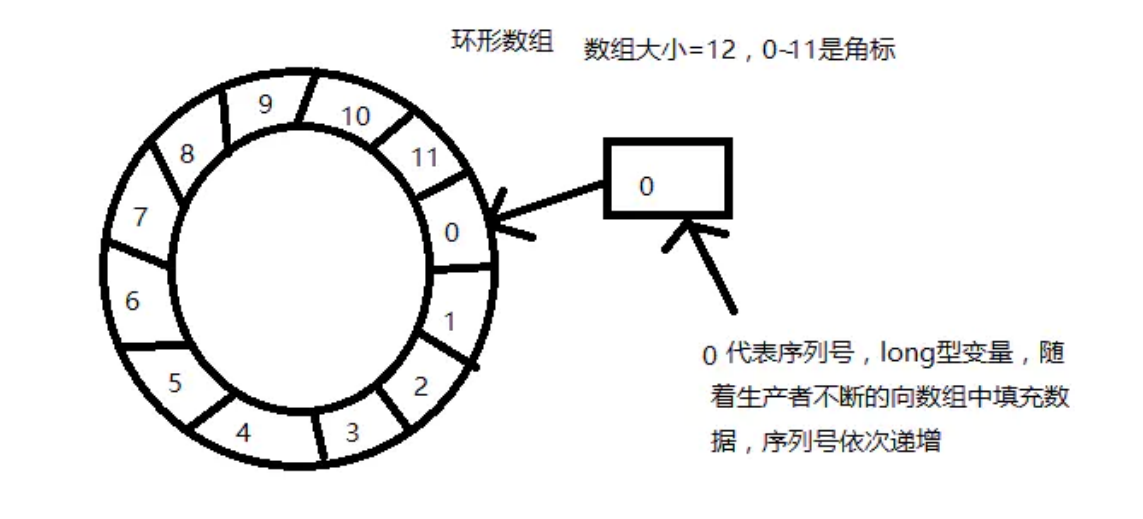
数组长度2^n，通过位运算，加快定位的速度。下标采取递增的形式。不用担心index溢出的问题。index是long类型，即使100万QPS的处理速度，也需要30万年才能用完。

无锁设计：

每个生产者或者消费者线程，会先申请可以操作的元素在数组中的位置，申请到之后，直接在该位置写入或者读取数据。

**数据结构**

框架使用RingBuffer来作为队列的数据结构，RingBuffer就是一个可自定义大小的环形数组。除数组外还有一个序列号(sequence)，用以指向下一个可用的元素，供生产者与消费者使用。原理图如下所示



**Sequence**

mark：Disruptor通过顺序递增的序号来编号管理通过其进行交换的数据（事件），对

数据(事件)的处理过程总是沿着序号逐个递增处理。

**数组+序列号设计的优势是什么呢？**

回顾一下我们讲HashMap时，在知道索引(index)下标的情况下，存与取数组上的元素

时间复杂度只有O(1)，而这个index我们可以通过序列号与数组的长度取模来计算得出，

index=sequence % table.length。当然也可以用位运算来计算效率更高，此时

table.length必须是2的幂次方(原理前面讲过)。

**概念与作用**

RingBuffer——Disruptor底层数据结构实现，核心类，是线程间交换数据的中转地；

Sequencer——序号管理器，生产同步的实现者，负责消费者/生产者各自序

号、序号栅栏的管理和协调,Sequencer有单生产者,多生产者两种不同的模式,里面实

现了各种同步的算法；

Sequence——序号，声明一个序号，用于跟踪ringbuffer中任务的变化和消费者的消费情况，disruptor里面大部分的并发代码都是通过对Sequence的值同步修改实现的,而非锁,这是disruptor高性能的一个主要原因；

SequenceBarrier——序号栅栏，管理和协调生产者的游标序号和各个消费者的序号，确保生产者不会覆盖消费者未来得及处理的消息，确保存在依赖的消费者之间能够按照正确的顺序处理， Sequence Barrier是由Sequencer创建的,并被Processor持有；

EventProcessor——事件处理器，监听RingBuffer的事件，并消费可用事件，从RingBuffer读取的事件会交由实际的生产者实现类来消费；它会一直侦听下一个可用的序号，直到该序号对应的事件已经准备好。

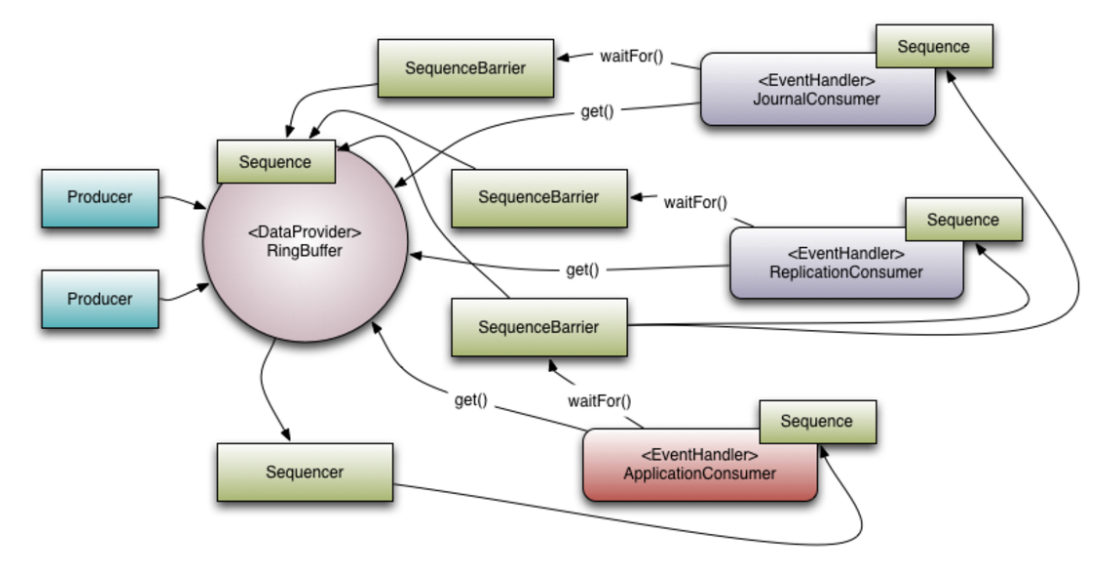
EventHandler——业务处理器，是实际消费者的接口，完成具体的业务逻辑实现，第三方实现该接口；代表着消费者。

Producer——生产者接口，第三方线程充当该角色，producer向RingBuffer写

入事件。

Wait Strategy：Wait Strategy决定了一个消费者怎么等待生产者将事件

（Event）放入Disruptor中。



**等待策略**

BlockingWaitStrategy

Disruptor的默认策略是BlockingWaitStrategy。在BlockingWaitStrategy内部是使用锁和condition来控制线程的唤醒。BlockingWaitStrategy是最低效的策略，但其对CPU

的消耗最小并且在各种不同部署环境中能提供更加一致的性能表现。

SleepingWaitStrategy

SleepingWaitStrategy 的性能表现跟 BlockingWaitStrategy 差不多，对 CPU 的消耗也类似，但其对生产者线程的影响最小，通过使用LockSupport.parkNanos(1)来实现循环等待。一般来说Linux系统会暂停一个线程约60µs，这样做的好处是，生产线程不需要采取任何其他行动就可以增加适当的计数器，也不需要花费时间信号通知条件变量。但是，在生产者线程和使用者线程之间移动事件的平均延迟会更高。它在不需要低延迟并且对生产线程的影响较小的情况最好。一个常见的用例是异步日志记录。

YieldingWaitStrategyYieldingWaitStrategy是可以使用在低延迟系统的策略之一。YieldingWaitStrategy

将自旋以等待序列增加到适当的值。在循环体内，将调用Thread.yield（），以允许其他排

队的线程运行。在要求极高性能且事件处理线数小于 CPU 逻辑核心数的场景中，推荐使用

此策略；例如，CPU开启超线程的特性。

BusySpinWaitStrategy

性能最好，适合用于低延迟的系统。在要求极高性能且事件处理线程数小于CPU逻辑核

心数的场景中，推荐使用此策略；例如，CPU开启超线程的特性。

**写数据**

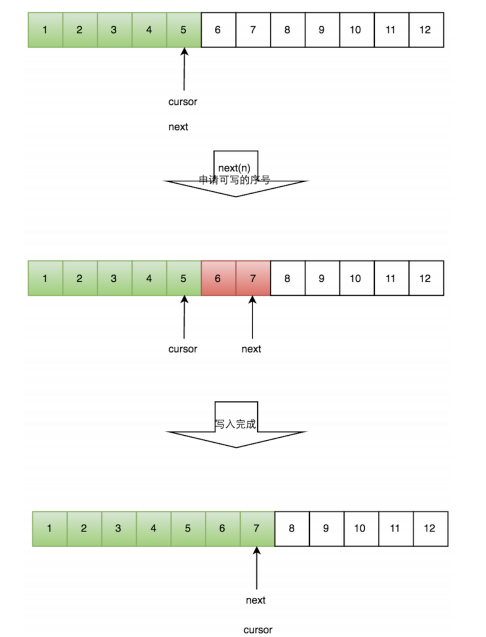
单线程写数据的流程：

1. 申请写入m个元素；

2. 若是有m个元素可以入，则返回最大的序列号。这儿主要判断是否会覆盖未读的元

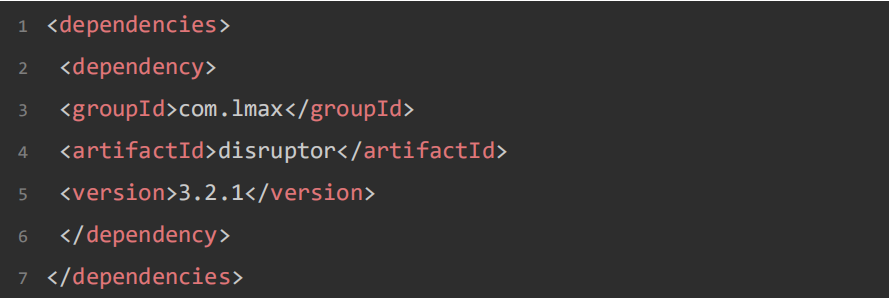
素；

3. 若是返回的正确，则生产者开始写入元素

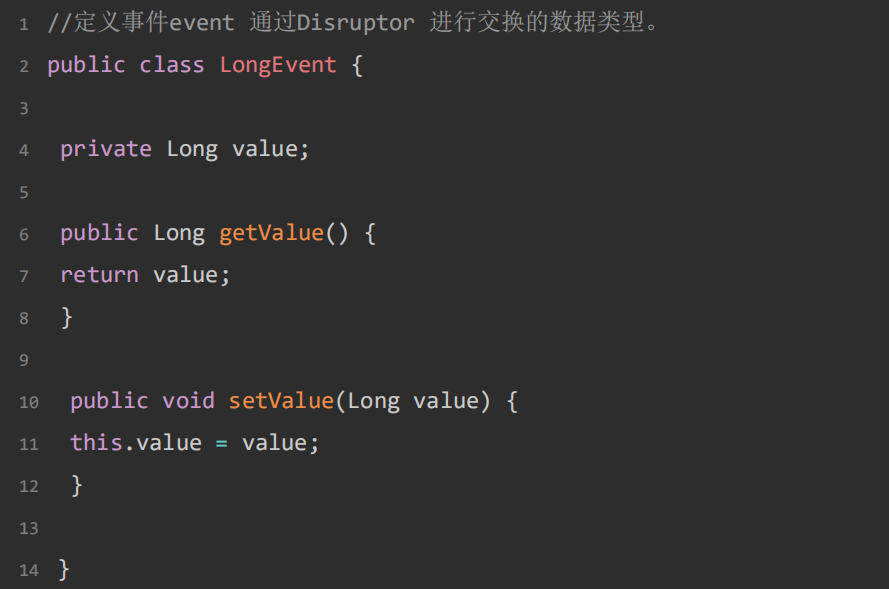


## 4.2 使用

1、引入依赖



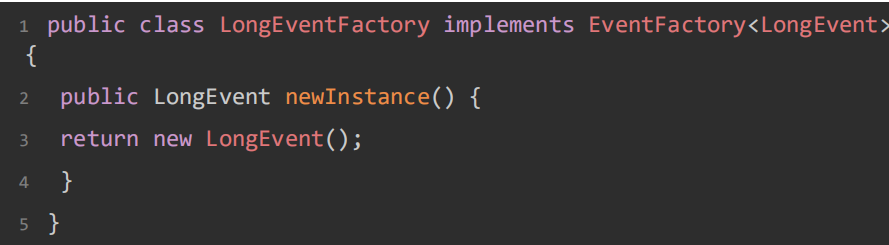
2、定义event



3、定义EventFactory

我们需要Disruptor为我们创建Event，所以这里我们需要定义事件工厂，实现框架定义的

接口

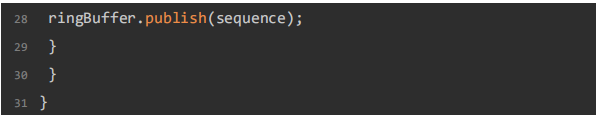


4、定义事件消费者



5、定义生产者





6、定义Main入口





