## Java 中的各种锁和 CAS + 面试题

如果说快速理解多线程有什么捷径的话，那本文介绍的各种锁无疑是其中之一，它不但为我们开发多线程程序提供理论支持，还是面试中经常被问到的核心面试题之一。因此下面就让我们一起深入地学习一下这些锁吧。

### 乐观锁和悲观锁

悲观锁和乐观锁并不是某个具体的“锁”而是一种并发编程的基本概念。乐观锁和悲观锁最早出现在数据库的设计当中，后来逐渐被 Java 的并发包所引入。

#### 悲观锁

悲观锁认为对于同一个数据的并发操作，一定是会发生修改的，哪怕没有修改，也会认为修改。因此对于同一个数据的并发操作，悲观锁采取加锁的形式。悲观地认为，不加锁的并发操作一定会出问题。

#### 乐观锁

乐观锁正好和悲观锁相反，它获取数据的时候，并不担心数据被修改，每次获取数据的时候也不会加锁，只是在更新数据的时候，通过判断现有的数据是否和原数据一致来判断数据是否被其他线程操作，如果没被其他线程修改则进行数据更新，如果被其他线程修改则不进行数据更新。

### 公平锁和非公平锁

根据线程获取锁的抢占机制，锁又可以分为公平锁和非公平锁。

#### 公平锁

公平锁是指多个线程按照申请锁的顺序来获取锁。

#### 非公平锁

非公平锁是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁。  
ReentrantLock 提供了公平锁和非公平锁的实现。

* 公平锁：new ReentrantLock(true)
* 非公平锁：new ReentrantLock(false)

如果构造函数不传任何参数的时候，默认提供的是非公平锁。

### 独占锁和共享锁

根据锁能否被多个线程持有，可以把锁分为独占锁和共享锁。

#### 独占锁

独占锁是指任何时候都只有一个线程能执行资源操作。

#### 共享锁

共享锁指定是可以同时被多个线程读取，但只能被一个线程修改。比如 Java 中的 ReentrantReadWriteLock 就是共享锁的实现方式，它允许一个线程进行写操作，允许多个线程读操作。

ReentrantReadWriteLock 共享锁演示代码如下：

public class ReadWriteLockTest {  
 public static void main(String[] args) throws InterruptedException {  
 final MyReadWriteLock rwLock = new MyReadWriteLock();  
 // 创建读锁 r1 和 r2  
 Thread r1 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 rwLock.read();  
 }  
 }, "r1");  
 Thread r2 = new Thread(new Runnable() {  
 @Override  
 public void run() {  
 rwLock.read();  
 }  
 }, "r2");  
 r1.start();  
 r2.start();  
 // 等待同时读取线程执行完成  
 r1.join();  
 r2.join();  
 // 开启写锁的操作  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 rwLock.write();  
 }  
 }, "w1").start();  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 rwLock.write();  
 }  
 }, "w2").start();  
 }  
 static class MyReadWriteLock {  
 ReadWriteLock lock = new ReentrantReadWriteLock();  
 public void read() {  
 try {  
 lock.readLock().lock();  
 System.out.println("读操作，进入 | 线程：" + Thread.currentThread().getName());  
 Thread.sleep(3000);  
 System.out.println("读操作，退出 | 线程：" + Thread.currentThread().getName());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.readLock().unlock();  
 }  
 }  
 public void write() {  
 try {  
 lock.writeLock().lock();  
 System.out.println("写操作，进入 | 线程：" + Thread.currentThread().getName());  
 Thread.sleep(3000);  
 System.out.println("写操作，退出 | 线程：" + Thread.currentThread().getName());  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } finally {  
 lock.writeLock().unlock();  
 }  
 }  
 }  
}

以上程序执行结果如下：

读操作，进入 | 线程：r1

读操作，进入 | 线程：r2

读操作，退出 | 线程：r1

读操作，退出 | 线程：r2

写操作，进入 | 线程：w1

写操作，退出 | 线程：w1

写操作，进入 | 线程：w2

写操作，退出 | 线程：w2

### 可重入锁

可重入锁指的是该线程获取了该锁之后，可以无限次的进入该锁锁住的代码。

### 自旋锁

自旋锁是指尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取锁，这样的好处是减少线程上下文切换的消耗，缺点是循环会消耗 CPU。

### CAS 与 ABA

CAS（Compare and Swap）比较并交换，是一种乐观锁的实现，是用非阻塞算法来代替锁定，其中 java.util.concurrent 包下的 AtomicInteger 就是借助 CAS 来实现的。  
但 CAS 也不是没有任何副作用，比如著名的 ABA 问题就是 CAS 引起的。

#### ABA 问题描述

老王去银行取钱，余额有 200 元，老王取 100 元，但因为程序的问题，启动了两个线程，线程一和线程二进行比对扣款，线程一获取原本有 200 元，扣除 100 元，余额等于 100 元，此时阿里给老王转账 100 元，于是启动了线程三抢先在线程二之前执行了转账操作，把 100 元又变成了 200 元，而此时线程二对比自己事先拿到的 200 元和此时经过改动的 200 元值一样，就进行了减法操作，把余额又变成了 100 元。这显然不是我们要的正确结果，我们想要的结果是余额减少了 100 元，又增加了 100 元，余额还是 200 元，而此时余额变成了 100 元，显然有悖常理，这就是著名的 ABA 的问题。

执行流程如下。

* 线程一：取款，获取原值 200 元，与 200 元比对成功，减去 100 元，修改结果为 100 元。
* 线程二：取款，获取原值 200 元，阻塞等待修改。
* 线程三：转账，获取原值 100 元，与 100 元比对成功，加上 100 元，修改结果为 200 元。
* 线程二：取款，恢复执行，原值为 200 元，与 200 元对比成功，减去 100 元，修改结果为 100 元。

最终的结果是 100 元。

#### ABA 问题的解决

常见解决 ABA 问题的方案加版本号，来区分值是否有变动。以老王取钱的例子为例，如果加上版本号，执行流程如下。

* 线程一：取款，获取原值 200\_V1，与 200\_V1 比对成功，减去 100 元，修改结果为 100\_V2。
* 线程二：取款，获取原值 200\_V1 阻塞等待修改。
* 线程三：转账，获取原值 100\_V2，与 100\_V2 对比成功，加 100 元，修改结果为 200\_V3。
* 线程二：取款，恢复执行，原值 200\_V1 与现值 200\_V3 对比不相等，退出修改。

最终的结果为 200 元，这显然是我们需要的结果。  
在程序中，要怎么解决 ABA 的问题呢？  
在 JDK 1.5 的时候，Java 提供了一个 AtomicStampedReference 原子引用变量，通过添加版本号来解决 ABA 的问题，具体使用示例如下：

String name = "老王";  
String newName = "Java";  
AtomicStampedReference<String> as = new AtomicStampedReference<String>(name, 1);  
System.out.println("值：" + as.getReference() + " | Stamp：" + as.getStamp());  
as.compareAndSet(name, newName, as.getStamp(), as.getStamp() + 1);  
System.out.println("值：" + as.getReference() + " | Stamp：" + as.getStamp());

以上程序执行结果如下：

值：老王 | Stamp：1

值：Java | Stamp：2

### 相关面试题

#### 1.synchronized 是哪种锁的实现？为什么？

答：synchronized 是悲观锁的实现，因为 synchronized 修饰的代码，每次执行时会进行加锁操作，同时只允许一个线程进行操作，所以它是悲观锁的实现。

#### 2.new ReentrantLock() 创建的是公平锁还是非公平锁？

答：非公平锁，查看 ReentrantLock 的实现源码可知。

/\*\*  
 \* Creates an instance of {@code ReentrantLock}.  
 \* This is equivalent to using {@code ReentrantLock(false)}.  
 \*/  
public ReentrantLock() {  
 sync = new NonfairSync();  
}

#### 3.synchronized 使用的是公平锁还是非公平锁？

答：synchronized 使用的是非公平锁，并且是不可设置的。这是因为非公平锁的吞吐量大于公平锁，并且是主流操作系统线程调度的基本选择，所以这也是 synchronized 使用非公平锁原由。

#### 4.为什么非公平锁吞吐量大于公平锁？

答：比如 A 占用锁的时候，B 请求获取锁，发现被 A 占用之后，堵塞等待被唤醒，这个时候 C 同时来获取 A 占用的锁，如果是公平锁 C 后来者发现不可用之后一定排在 B 之后等待被唤醒，而非公平锁则可以让 C 先用，在 B 被唤醒之前 C 已经使用完成，从而节省了 C 等待和唤醒之间的性能消耗，这就是非公平锁比公平锁吞吐量大的原因。

#### 5.volatile 的作用是什么？

答：volatile 是 Java 虚拟机提供的最轻量级的同步机制。  
当变量被定义成 volatile 之后，具备两种特性：

* 保证此变量对所有线程的可见性，当一条线程修改了这个变量的值，修改的新值对于其他线程是可见的（可以立即得知的）；
* 禁止指令重排序优化，普通变量仅仅能保证在该方法执行过程中，得到正确结果，但是不保证程序代码的执行顺序。

#### 6.volatile 对比 synchronized 有什么区别？

答：synchronized 既能保证可见性，又能保证原子性，而 volatile 只能保证可见性，无法保证原子性。比如，i++ 如果使用 synchronized 修饰是线程安全的，而 volatile 会有线程安全的问题。

#### 7.CAS 是如何实现的？

答： CAS（Compare and Swap）比较并交换，CAS 是通过调用 JNI（Java Native Interface）的代码实现的，比如，在 Windows 系统 CAS 就是借助 C 语言来调用 CPU 底层指令实现的。

#### 8.CAS 会产生什么问题？应该怎么解决？

答：CAS 是标准的乐观锁的实现，会产生 ABA 的问题（详见正文）。  
ABA 通常的解决办法是添加版本号，每次修改操作时版本号加一，这样数据对比的时候就不会出现 ABA 的问题了。

#### 9.以下说法错误的是？

A：独占锁是指任何时候都只有一个线程能执行资源操作  
B：共享锁指定是可以同时被多个线程读取和修改  
C：公平锁是指多个线程按照申请锁的顺序来获取锁  
D：非公平锁是指多个线程获取锁的顺序并不是按照申请锁的顺序，有可能后申请的线程比先申请的线程优先获取锁  
答：B  
题目解析：共享锁指定是可以同时被多个线程读取，但只能被一个线程修改。

### 总结

本文介绍了 Java 中各种锁，明白了 Java 程序中比较常用的为非公平锁而非公平锁，原因在于非公平锁的吞吐量要更大，并且发生线程“饥饿”的情况很少，是风险远小于收益的事所以可以广而用之。又重点介绍了 CAS 和著名的 ABA 的问题，以及解决 ABA 的常见手段：添加版本号，可以通过 Java 自身提供的 AtomicStampedReference（原子引用变量）来解决 ABA 的问题，至此我们对 Java 多线程的了解又向前迈了一大步。

[点击此处下载本文源码](https://github.com/vipstone/java-%20interview/tree/master/interview-code/src/main/java/com/interview)

更多java学习资料如下：

[Java高级架构师课程(总共122门课程，1460GB)](https://www.consultdog.com/index.php/front/44/)

[最强java面试视频课程(21门课程，126GB)](https://www.consultdog.com/index.php/front/78/)