1、面试题

java里玩儿悲观锁和乐观锁一般怎么玩儿？synchronized相当于是悲观锁，CAS相当于是乐观锁。知道CAS是什么吗？CAS是如何实现的？

2、面试官心里分析

这个高级点的面试，肯定会问CAS，还是比较重要的

3、面试题剖析

悲观锁：我现在要操作一个共享数据，我很悲观，我认为我操作的过程中，一定会被人给修改，会导致数据错误；我在操作这个数据之前，先给这个数据加了一把锁，synchronized，在我操作这个数据的期间，就只能是我来操作，其他任何人都操作不了。

乐观锁：我感觉在我操作这个数据的过程中，应该不会被人给修改。我先修改吗，然后修改完之后要设置这个变量的最新的值，此时我会对比一下，当前的这个值，是不是跟我在操作前看到的这个值是一样的，如果是一样的，那么说明可能就没有被人给修改过，如果没有被人修改过，那么我就可以来设置最新的值。

CAS，Compare and Swap，就是比较和交换。java并发包借助CAS实现了乐观锁的思想，synchronized是悲观锁思想。

CAS会操作3个数字，当前内存中的值，旧的预期值，新的修改值，只有当旧的预期值跟内存中的值一样的时候，才会将内存中的值修改为新的修改值。举个例子吧，比如int a = 3，这是内存中的当前值，然后你CAS（3, 5），第一个是旧的预期值，如果3和a是一样的，那么就将a修改为5。

public class Test {

private int i = 0; // 假设i是一个类中的共享变量

public synchronized void incr() {

i++; // 不是线程安全的

}

}

可能2个线程进来，执行incr()方法，期望的是i -> 2，但是其实可能会变成1

public class Test {

private AtomicInteger i = new AtomicInteger(0);

public void incr() {

int next = i.getAndIncrement();

}

}

无论是多少个线程并发执行这段代码，都是ok的，2个线程过来，一定是i -> 2，不可能是1，CAS这个东西，可以保证一个变量的操作都是原子的

CAS里的经典用法就是Atomic系列类，比如说AtomicInteger

还是挺常用的，常见于什么呢？常见于内存计数，比如说你要在内存里维护一个变量，记录每个请求过来的一个次数这样子，你就可以用这个AtomicInteger，可以安全的去给他累加对应的值，原子的

public final int incrementAndGet() {

for (;;) {

int current = get(); // 先拿到i当前的值，0

int next = current + 1; // 对i加1 -> 1

if (compareAndSet(current, next))

// 看一下，i这个变量，当前的值是不是0，如果是0的话，就认为在上面两行代码执行的时候，没有人修改过i这个变量的值，所以就可以将i的值设置为1；但是如果i变量的值已经变成1了，会发现跟自己期望的0，不是一个值，说明上面两行代码执行期间，有人已经修改过了i的值，所以你就不能再次将这个1这个值设置给i了

// 此时compareAndSet()返回一个false，直接进入下一轮循环

// 干了一样的事儿，先获取i当前的值，比如说现在变成了1这个值了

// 再次将i加1 -> 2

// 再次执行compareAndSet(1, 2)，看一下i变量当前的值是不是1，如果是，就证明这个期间没人修改过这个值，就将最新的值2设置给i变量

// 然后compareAndSet()方法就返回true

// return next;，直接跳出一个死循环

// 对于调用者来说，可以拿到本次累加完以后的一个当前值

return next;

}

}

就是实现一个++i的效果，对一个数字累加1后返回，如何实现原子的呢？

就是获取这个数字当前的值，然后加1之后，用compareAndSet(current, next)方法来设置，这个方法会比较内存中的当前值是不是current，如果是才会设置为next也就是加1后的数字，否则的话，就会再次重试，for(;;)就是个无限循环的操作

public final boolean compareAndSet(int expect, int update) {

return unsafe.compareAndSwapInt(this, valueOffset, expect, update);

}

这个compareAndSet()方法，其实是用了cpu的指令，JNI本地方法调用实现的，现代cpu的compareAndSwapInt的指令，就是可以实现原子的操作，他会保证说，当前时刻就一个线程可以针对个数字，比较当前值，然后设置最新值，如果当前值不一致，那么会返回false

其实吧，这里比较关键的一点就是cpu的compareAndSwap操作的原理是啥，以CPU（Intel X86）来举个例子。这块底层指令，会根据当前处理器类型，来决定要不要对一个cmpxchg指令加lock前缀，如果是单处理器，就不要加，因为自动保证顺序；但是如果是多处理器，就加个lock。intel对lock的定义，就是说加了lock之后，就会自动锁掉一块内存区域，然后同一时间只有一个处理器可以读写这块内存区域，其他处理器就不行了。

而且intel对这个lock做了优化，以前都是走一种总线锁，就是一旦锁住之后，只有一个处理器可以操作内存，别的处理器完全不能操作内存，保证说，只有一个cpu可以操作内存，其他cpu都不能操作。相当于是在cpu层面干了类似synchronized这个事儿，这个效率太低了，所以现在现代cpu都不会干这个事儿。整个内存可能包含了很多的数据，就导致同一时间只有一个cpu可以操作内存，这个效率实在是太低了。

但是后来用了缓存锁优化，缓存一致性协议，其实就是处理器对自己内部的缓存锁了，有多个cpu，其中一个cpu就会锁定i变量对应的一行内存，其他cpu就不能对i这个变量进行操作了，同一时间就只能是一个cpu对i变量查看值和设置直；然后就一个内存地址，别的处理器不能操作，但是别的处理器可以处理别的内存地址，其实就是降低了锁的粒度。

CAS其实有3个缺点：

1、ABA问题：如果某个值一开始是A，后来变成了B，然后又变成了A，你本来期望的是值如果是第一个A才会设置新值，结果第二个A一比较也ok，也设置了新值，跟期望是不符合的。所以atomic包里有AtomicStampedReference类，就是会比较两个值的引用是否一致，如果一致，才会设置新值

假设一开始变量i = 1，你先获取这个i的值是1，然后累加了1，变成了2

但是在此期间，别的线程将i -> 1 -> 2 -> 3 -> 1

这个期间，这个值是被人改过的，只不过最后将这个值改成了跟你最早看到的值一样的值

结果你后来去compareAndSet的时候，会发现这个i还是1，就将它设置成了2，就设置成功了

说实话，用AtomicInteger，常见的是计数，所以说一般是不断累加的，所以ABA问题比较少见

2、无限循环问题：大家看源码就知道Atomic类设置值的时候会进入一个无限循环，只要不成功，就不停循环再次尝试，这个在高并发修改一个值的时候其实挺常见的，比如你用AtomicInteger在内存里搞一个原子变量，然后高并发下，多线程频繁修改，其实可能会导致这个compareAndSet()里要循环N次才设置成功，所以还是要考虑到的。

3、多变量原子问题：一般的AtomicInteger，只能保证一个变量的原子性，但是如果多个变量呢？你可以用AtomicReference，这个是封装自定义对象的，多个变量可以放一个自定义对象里，然后他会检查这个对象的引用是不是一个。

853769620