ArrayList总结（特点：扩容1.5倍、Object数组实现、序列化：readObject和writeObject）

1、继承：AbstractList类

2、实现：

List

RandomAccess（标记型的接口，只是说明实现类可以随机访问）

Cloneable（标记型的接口，说明对象可以被克隆，调用Object.clone()不会抛出CloneNotSupportedException异常

java.io.Serializable(可以被序列化)

3、重要的成员变量

静态常量：

Ⅰ、private static final int DEFAULT\_CAPACITY=10//默认容量

Ⅱ、private static final Object[] EMPTY\_ELEMENTDATA= {}

Ⅲ、private static final Object[] DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA = {}

Ⅳ、transient Object[] elementData;

①transient避免序列化，所以这个字段的生命周期仅存于调用者的内存中，不会写到磁盘里持久化。

②非私有化以简化嵌套类访问

V、private int size//实际大小

Ⅵ、private static final int MAX\_ARRAY\_SIZE = Integer.MAX\_VALUE – 8//最大数组容量

Ⅶ、modCount：记录着集合的修改次数

4、重要的方法

Ⅰ、public Object[] toArray()

Ⅱ、private void writeObject(java.io.ObjectOutputStream s)

Ⅲ、private void readObject(java.io.ObjectInputStream s)

5、遍历方式

Ⅰ、迭代器或forEach

Ⅱ、get(index)

Ⅲ、for:

Synchronized关键字（wait/notify等方法也依赖于monitor对象）

一、作用

1、确保线程互斥的访问同步代码

2、保证共享变量的修改能够及时可见

3、有效解决重排序问题

二、用法

1、修饰普通方法：对同一个对象（的方法）进行同步（可与其他对象并行）。但是其他线程还是可以访问该实例对象的其他非synchronized方法

2、修饰静态方法：对同一个类（的静态方法）进行同步。对静态方法的同步本质上是对类的同步，因为静态方法本质上是属于类的方法，而不是对象上的方法。（可与其他类并行）

3、修饰代码块：对同一个对象或类（的代码块）进行同步

三、原理：

每个对象有一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1、如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。

2、如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1.（可重入）

3.如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

注意：

1、代码块（monitorenter和monitorexit）

2、方法级的同步是隐式，即无需通过字节码指令来控制的。ACC\_SYNCHRONIZED：该标识指明了方法是否为一个同步方法

四、低效的原因：依赖于底层的操作系统的Mutex Lock实现的锁（重量级锁）

监视器锁本质又是依赖于底层的操作系统的Mutex Lock来实现的。而操作系统实现线程之间的切换这就需要从用户态转换到核心态，这个成本非常高，状态之间的转换需要相对比较长的时间。

五、等待唤醒机制

指notify/notifyAll和wait方法，在使用这3个方法时，必须处于synchronized代码块或者synchronized方法中，否则就会抛出IllegalMonitorStateException异常，这是因为调用这几个方法前必须拿到当前对象的监视器monitor对象，也就是说notify/notifyAll和wait方法依赖于monitor对象，在前面的分析中，我们知道monitor 存在于对象头的Mark Word 中(存储monitor引用指针)，而synchronized关键字可以获取 monitor ，这也就是为什么notify/notifyAll和wait方法必须在synchronized代码块或者synchronized方法调用的原因。

六、锁的四种状态（锁的升级是单向的，只能从低到高升级）

1、无锁状态

2、偏向锁

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。

3、轻量级锁00

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。

4、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。

5、重量级锁10

6、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。