# 面试题

### 如何用数组实现队列？

用数组实现队列时要注意 **溢出** 现象，这时我们可以采用循环数组的方式来解决，即将数组收尾相接。使用front指针指向队列首位，tail指针指向队列末位。

### 内部类访问局部变量的时候，为什么变量必须加上final修饰？

因为生命周期不同。局部变量在方法结束后就会被销毁，但内部类对象并不一定，这样就会导致内部类引用了一个不存在的变量。

所以编译器会在内部类中生成一个局部变量的拷贝，这个拷贝的生命周期和内部类对象相同，就不会出现上述问题。

但这样就导致了其中一个变量被修改，两个变量值可能不同的问题。为了解决这个问题，编译器就要求局部变量需要被final修饰，以保证两个变量值相同。

在JDK8之后，编译器不要求内部类访问的局部变量必须被final修饰，但局部变量值不能被修改（无论是方法中还是内部类中），否则会报编译错误。利用javap查看编译后的字节码可以发现，编译器已经加上了final。

### long s = 499999999 \* 499999999 在上面的代码中，s的值是多少？

根据代码的计算结果，s的值应该是-1371654655，**这是由于Java中右侧值的计算默认是**int类型。

### NIO相关，Channels、Buffers、Selectors

NIO(Non-blocking IO)为所有的原始类型提供(Buffer)缓存支持，字符集编码解码解决方案。 Channel ：一个新的原始I/O 抽象。 支持锁和内存映射文件的文件访问接口。提供多路(non-bloking) 非阻塞式的高伸缩性网络I/O 。

|  |  |
| --- | --- |
| IO | NIO |
| 面向流 | 面向缓冲 |
| 阻塞IO | 非阻塞IO |
| 无 | 选择器 |

##### 流与缓冲

Java NIO和IO之间第一个最大的区别是，**IO是面向流的，NIO是面向缓冲区的**。 Java IO面向流意味着每次从流中读一个或多个字节，直至读取所有字节，它们没有被缓存在任何地方。此外，它不能前后移动流中的数据。如果需要前后移动从流中读取的数据，需要先将它缓存到一个缓冲区。

Java NIO的缓冲导向方法略有不同。**数据读取到一个它稍后处理的缓冲区，需要时可在缓冲区中前后移动。这就增加了处理过程中的灵活性**。但是，还需要检查是否该缓冲区中包含所有您需要处理的数据。而且，需确保当更多的数据读入缓冲区时，不要覆盖缓冲区里尚未处理的数据。

##### 阻塞与非阻塞IO

Java IO的各种流是阻塞的。这意味着，当一个线程调用read() 或 write()时，该线程被阻塞，直到有一些数据被读取，或数据完全写入。该线程在此期间不能再干任何事情了。 **Java NIO的非阻塞模式，是线程向某通道发送请求读取数据，仅能得到目前可用的数据，如果目前没有数据可用时，就什么都不会获取，当然它不会保持线程阻塞。所以直至数据变的可以读取之前，该线程可以继续做其他的事情**。 非阻塞写也是如此。所以一个单独的线程现在可以管理多个输入和输出通道。

##### 选择器（Selectors）

Java NIO 的 **选择器允许一个单独的线程来监视多个输入通道**，你可以注册多个通道使用一个选择器，然后使用一个单独的线程来“选择”通道：这些通道里已经有可以处理的输入，或者选择已准备写入的通道。这种选择机制，使得一个单独的线程很容易来管理多个通道。

### 反射的用途

Java反射机制可以让我们在编译期(Compile Time)之外的运行期(Runtime)检查类，接口，变量以及方法的信息。反射还可以让我们在运行期实例化对象，调用方法，通过调用get/set方法获取变量的值。同时我们也可以通过反射来获取泛型信息，以及注解。还有更高级的应用--动态代理和动态类加载（ClassLoader.loadclass()）。

下面列举一些比较重要的方法：

* getFields：获取所有 public 的变量。
* getDeclaredFields：获取所有包括 private , protected 权限的变量。
* setAccessible：设置为 true 可以跳过Java权限检查，从而访问private权限的变量。
* getAnnotations：获取注解，可以用在类和方法上。

获取方法的泛型参数：

method = Myclass.class.getMethod("setStringList", List.class);

Type[] genericParameterTypes = method.getGenericParameterTypes();

for(Type genericParameterType : genericParameterTypes){
 if(genericParameterType instanceof ParameterizedType){
 ParameterizedType aType = (ParameterizedType) genericParameterType;
 Type[] parameterArgTypes = aType.getActualTypeArguments();
 for(Type parameterArgType : parameterArgTypes){
 Class parameterArgClass = (Class) parameterArgType;
 System.out.println("parameterArgClass = " + parameterArgClass);
 }
 }
}

动态代理：

//Main.java
public static void main(String[] args) {
 HelloWorld helloWorld=new HelloWorldImpl();
 InvocationHandler handler=new HelloWorldHandler(helloWorld);

 //创建动态代理对象
 HelloWorld proxy=(HelloWorld)Proxy.newProxyInstance(
 helloWorld.getClass().getClassLoader(),
 helloWorld.getClass().getInterfaces(),
 handler);
 proxy.sayHelloWorld();
}

//HelloWorldHandler.java
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
 Object result = null;
 //调用之前
 doBefore();
 //调用原始对象的方法
 result=method.invoke(obj, args);
 //调用之后
 doAfter();
 return result;
}

通过反射获取方法注解的参数：

Class aClass = TheClass.class;
Annotation[] annotations = aClass.getAnnotations();

for(Annotation annotation : annotations){
 if(annotation instanceof MyAnnotation){
 MyAnnotation myAnnotation = (MyAnnotation) annotation;
 System.out.println("name: " + myAnnotation.name());
 System.out.println("value: " + myAnnotation.value());
 }
}

### 非静态内部类能定义静态方法吗？

public class OuterClass{
 private static float f = 1.0f;

 class InnerClass{
 public static float func(){return f;}
 }
}

以上代码会出现编译错误，因为只有静态内部类才能定义静态方法。

### Lock 和 Synchronized 有什么区别？

1. 使用方法的区别
	* **Synchronized**：在需要同步的对象中加入此控制，synchronized可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。
	* **Lock**：需要显示指定起始位置和终止位置。一般使用ReentrantLock类做为锁，多个线程中必须要使用一个ReentrantLock类做为对象才能保证锁的生效。且在加锁和解锁处需要通过lock()和unlock()显示指出。所以一般会在finally块中写unlock()以防死锁。
2. 性能的区别
* synchronized是托管给JVM执行的，而lock是java写的控制锁的代码。在Java1.5中，synchronize是性能低效的。因为这是一个重量级操作，需要调用操作接口，导致有可能加锁消耗的系统时间比加锁以外的操作还多。相比之下使用Java提供的Lock对象，性能更高一些。但是到了Java1.6，发生了变化。synchronize在语义上很清晰，可以进行很多优化，有适应自旋，锁消除，锁粗化，轻量级锁，偏向锁等等。导致在Java1.6上synchronize的性能并不比Lock差。
	+ **Synchronized**：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 **其他线程只能依靠阻塞来等待线程释放锁**。而在CPU转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起CPU频繁的上下文切换导致效率很低。
	+ **Lock**：用的是乐观锁方式。所谓乐观锁就是，**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**。乐观锁实现的机制就是CAS操作。我们可以进一步研究ReentrantLock的源代码，会发现其中比较重要的获得锁的一个方法是compareAndSetState。这里其实就是调用的CPU提供的特殊指令。
1. ReentrantLock：具有更好的可伸缩性：比如时间锁等候、可中断锁等候、无块结构锁、多个条件变量或者锁投票。

### float 变量如何与 0 比较？

folat类型的还有double类型的，**这些小数类型在趋近于0的时候直接等于0的可能性很小，一般都是无限趋近于0，因此不能用==来判断**。应该用|x-0|<err来判断，这里|x-0|表示绝对值，err表示限定误差。

//用程序表示就是

fabs(x) < 0.00001f

### 如何新建非静态内部类？

内部类在声明的时候必须是 Outer.Inner a，就像int a 一样，至于静态内部类和非静态内部类new的时候有点区别：

* Outer.Inner a = new Outer().new Inner()（非静态，先有Outer对象才能 new 内部类）
* Outer.Inner a = new Outer.Inner()（静态内部类）

### Java标识符命名规则

可以包含：字母、数字、$、\_(下划线)，不可用数字开头，不能是 Java 的关键字和保留字。

### 你知道哪些JDK中用到的设计模式？

* 装饰模式：java.io
* 单例模式：Runtime类
* 简单工厂模式：Integer.valueOf方法
* 享元模式：String常量池、Integer.valueOf(int i)、Character.valueOf(char c)
* 迭代器模式：Iterator
* 职责链模式：ClassLoader的双亲委派模型
* 解释器模式：正则表达式java.util.regex.Pattern

### ConcurrentHashMap如何保证线程安全

JDK 1.7及以前：

ConcurrentHashMap允许多个修改操作并发进行，其关键在于使用了锁分离技术。它使用了多个锁来控制对hash表的不同部分进行的修改。ConcurrentHashMap内部使用段(Segment)来表示这些不同的部分，每个段其实就是一个小的hash table，它们有自己的锁。只要多个修改操作发生在不同的段上，它们就可以并发进行。

详细参考：

<http://www.cnblogs.com/ITtangtang/p/3948786.html>

[http://qifuguang.me/2015/09/10/[Java并发包学习八]深度剖析ConcurrentHashMap/](http://qifuguang.me/2015/09/10/%5BJava%E5%B9%B6%E5%8F%91%E5%8C%85%E5%AD%A6%E4%B9%A0%E5%85%AB%5D%E6%B7%B1%E5%BA%A6%E5%89%96%E6%9E%90ConcurrentHashMap/)

JDK 1.8：

Segment虽保留，但已经简化属性，仅仅是为了兼容旧版本。

插入时使用CAS算法：unsafe.compareAndSwapInt(this, valueOffset, expect, update)。 CAS(Compare And Swap)意思是如果valueOffset位置包含的值与expect值相同，则更新valueOffset位置的值为update，并返回true，否则不更新，返回false。插入时不允许key或value为null

与Java8的HashMap有相通之处，底层依然由“数组”+链表+红黑树；

底层结构存放的是TreeBin对象，而不是TreeNode对象；

CAS作为知名无锁算法，那ConcurrentHashMap就没用锁了么？当然不是，当hash值与链表的头结点相同还是会synchronized上锁，锁链表。

### i++在多线程环境下是否存在问题，怎么解决？

虽然递增操作++i是一种紧凑的语法，使其看上去只是一个操作，但这个操作并非原子的，因而它并不会作为一个不可分割的操作来执行。实际上，它包含了三个独立的操作：读取count的值，将值加1，然后将计算结果写入count。这是一个“读取 - 修改 - 写入”的操作序列，并且其结果状态依赖于之前的状态。所以在多线程环境下存在问题。

要解决自增操作在多线程环境下线程不安全的问题，可以选择使用Java提供的原子类，如AtomicInteger或者使用synchronized同步方法。

### new与newInstance()的区别

* new是一个关键字，它是调用new指令创建一个对象，然后调用构造方法来初始化这个对象，可以使用带参数的构造器
* newInstance()是Class的一个方法，在这个过程中，是先取了这个类的不带参数的构造器Constructor，然后调用构造器的newInstance方法来创建对象。

Class.newInstance不能带参数，如果要带参数需要取得对应的构造器，然后调用该构造器的Constructor.newInstance(Object ... initargs)方法

### 你了解哪些JDK1.8的新特性？

* 接口的默认方法和静态方法，JDK8允许我们给接口添加一个非抽象的方法实现，只需要使用default关键字即可。也可以定义被static修饰的静态方法。
* 对HashMap进行了改进，当单个桶的元素个数大于6时就会将实现改为红黑树实现，以避免构造重复的hashCode的攻击
* 多并发进行了优化。如ConcurrentHashMap实现由分段加锁、锁分离改为CAS实现。
* JDK8拓宽了注解的应用场景，注解几乎可以使用在任何元素上，并且允许在同一个地方多次使用同一个注解
* Lambda表达式

### 你用过哪些JVM参数？

* Xms 堆最小值
* Xmx 堆最大值
* Xmn: 新生代容量
* XX:SurvivorRatio 新生代中Eden与Surivor空间比例
* Xss 栈容量
* XX:PermSize 方法区初始容量
* XX:MaxPermSize 方法区最大容量
* XX:+PrintGCDetails 收集器日志参数

### 如何打破 ClassLoader 双亲委托？

重写loadClass()方法。

### hashCode() && equals()

hashcode() 返回该对象的哈希码值，支持该方法是为哈希表提供一些优点，例如，java.util.Hashtable 提供的哈希表。

在 Java 应用程序执行期间，在同一对象上多次调用 hashCode 方法时，必须一致地返回相同的整数，前提是对象上 equals 比较中所用的信息没有被修改（equals默认返回对象地址是否相等）。如果根据 equals(Object)方法，两个对象是相等的，那么在两个对象中的每个对象上调用 hashCode 方法都必须生成相同的整数结果。

以下情况不是必需的：如果根据 equals(java.lang.Object) 方法，两个对象不相等，那么在两个对象中的任一对象上调用 hashCode 方法必定会生成不同的整数结果。但是，**程序员应该知道，为不相等的对象生成不同整数结果可以提高哈希表的性能**。

实际上，由 Object 类定义的 hashCode 方法确实会针对不同的对象返回不同的整数。（**这一般是通过将该对象的内部地址转换成一个整数来实现的，但是 JavaTM 编程语言不需要这种实现技巧I**。）

* **hashCode的存在主要是用于查找的快捷性**，如 Hashtable，HashMap等，hashCode 是用来在散列存储结构中确定对象的存储地址的；
* 如果两个对象相同，就是适用于 equals(java.lang.Object) 方法，那么这两个对象的 hashCode 一定要相同；
* 如果对象的 equals 方法被重写，那么对象的 hashCode 也尽量重写，并且产生 hashCode 使用的对象，一定要和 equals 方法中使用的一致，否则就会违反上面提到的第2点；
* **两个对象的hashCode相同，并不一定表示两个对象就相同，也就是不一定适用于equals(java.lang.Object) 方法，只能够说明这两个对象在散列存储结构中，如Hashtable，他们“存放在同一个篮子里”**。

### Thread.sleep() & Thread.yield()

sleep()和yield()都会释放CPU。

sleep()使当前线程进入停滞状态，所以执行sleep()的线程在指定的时间内肯定不会执行；yield()只是使当前线程重新回到可执行状态，所以执行yield()的线程有可能在进入到可执行状态后马上又被执行。

sleep()可使优先级低的线程得到执行的机会，当然也可以让同优先级和高优先级的线程有执行的机会；yield()只能使同优先级的线程有执行的机会。