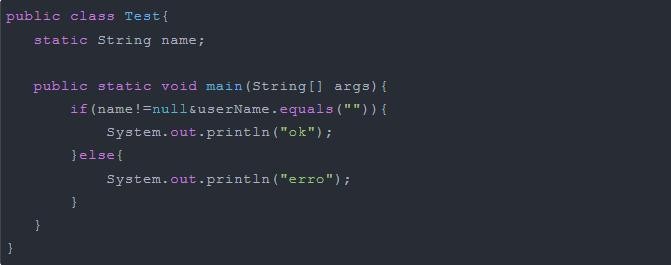
# 相关

## & 和 &&的区别

首先记住&是位操作，而&&是逻辑运算符。另外需要记住逻辑运算符具有短路特性，而&不 具备短路特性。





### 以上代码将会抛出空指针异常。

**一个 java 文件内部可以有类？(非内部类)**

只能有一个 public 公共类，但是可以有多个 default 修饰的类。

## 如何正确的退出多层嵌套循环？

使用标号和 break;

通过在外层循环中添加标识符

## 内部类的作用

内部类可以有多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独 立.在单个外围类当中，可以让多个内部类以不同的方式实现同一接口，或者继承同一个类. 创建内部类对象的时刻不依赖于外部类对象的创建。内部类并没有令人疑惑的”is-a”管系， 它就像是一个独立的实体。

内部类提供了更好的封装，除了该外围类，其他类都不能访问。

## final, finalize 和 finally 的不同之处

final 是一个修饰符，可以修饰变量、方法和类。如果 final 修饰变量，意味着该变量的值在 初始化后不能被改变。finalize 方法是在对象被回收之前调用的方法，给对象自己最后一个 复活的机会，但是什么时候调用 finalize 没有保证。finally 是一个关键字，与 try 和 catch 一起用于异常的处理。finally 块一定会被执行，无论在 try 块中是否有发生异常。

## clone()是哪个类的方法?

java.lang.Cloneable 是一个标示性接口，不包含任何方法，clone 方法在 object 类中定义。 并且需要知道 clone() 方法是一个本地方法，这意味着它是由 c 或 c++ 或 其他本地语言 实现的。

## 深拷贝和浅拷贝的区别是什么?

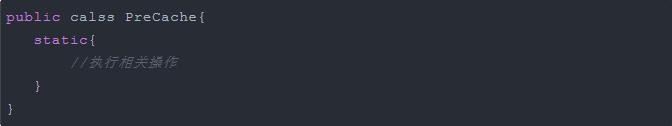
浅拷贝：被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用 仍然指向原来的对象。换言之，浅拷贝仅仅复制所考虑的对象，而不复制它所引用的对象。

深拷贝：被复制对象的所有变量都含有与原来的对象相同的值，而那些引用其他对象的变量 将指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深拷贝把要复制的 对象所引用的对象都复制了一遍。

## static 都有哪些用法?

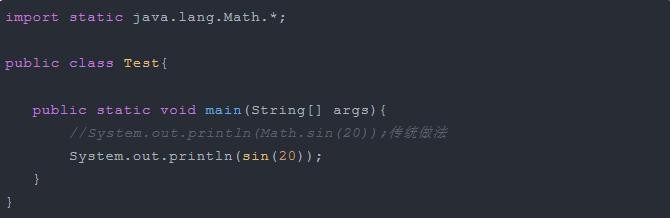
几乎所有的人都知道 static 关键字这两个基本的用法：静态变量和静态方法。也就是被 static 所修饰的变量/方法都属于类的静态资源，类实例所共享。

除了静态变量和静态方法之外，static 也用于静态块，多用于初始化操作：



此外 static 也多用于修饰内部类，此时称之为静态内部类。

最后一种用法就是静态导包，即 import static.import static 是在 JDK 1.5 之后引入的新特 性，可以用来指定导入某个类中的静态资源，并且不需要使用类名。资源名，可以直接使用 资源名，比如：



## final 有哪些用法

final 也是很多面试喜欢问的地方，能回答下以下三点就不错了： 1.被 final 修饰的类不可以被继承

2.被 final 修饰的方法不可以被重写

3.被 final 修饰的变量不可以被改变。如果修饰引用，那么表示引用不可变，引用指向的内容 可变。

4.被 final 修饰的方法，JVM 会尝试将其内联，以提高运行效率

5.被 final 修饰的常量，在编译阶段会存入常量池中。

回答出编译器对 final 域要遵守的两个重排序规则更好： 1.在构造函数内对一个 final 域的写入，与随后把这个被构造对象的引用赋值给一个引用变量, 这两个操作之间不能重排序。

2.初次读一个包含 final 域的对象的引用，与随后初次读这个 final 域,这两个操作之间不能重 排序。

# 进程,线程相关

## 说说进程，线程，协程之间的区别

简而言之，进程是程序运行和资源分配的基本单位，一个程序至少有一个进程，一个进程至 少有一个线程。进程在执行过程中拥有独立的内存单元，而多个线程共享内存资源，减少切 换次数，从而效率更高。线程是进程的一个实体，是 cpu 调度和分派的基本单位，是比程序 更小的能独立运行的基本单位。同一进程中的多个线程之间可以并发执行。

## 你了解守护线程吗？它和非守护线程有什么区别

程序运行完毕，jvm 会等待非守护线程完成后关闭，但是 jvm 不会等待守护线程。守护线程 最典型的例子就是 GC 线程。

## 什么是多线程上下文切换

多线程的上下文切换是指 CPU 控制权由一个已经正在运行的线程切换到另外一个就绪并等 待获取 CPU 执行权的线程的过程。

## 创建两种线程的方式?他们有什么区别?

通过实现 java.lang.Runnable 或者通过扩展 java.lang.Thread 类。 相比扩展 Thread， 实现

Runnable 接口可能更优.原因有二：

Java 不支持多继承。因此扩展 Thread 类就代表这个子类不能扩展其他类。而实现 Runnable 接口的类还可能扩展另一个类。

类可能只要求可执行即可，因此继承整个 Thread 类的开销过大。

## Thread 类中的 start()和 run()方法有什么区别?

start()方法被用来启动新创建的线程，而且 start()内部调用了 run()方法，这和直接调用 run() 方法的效果不一样。当你调用 run()方法的时候，只会是在原来的线程中调用，没有新的线 程启动，start()方法才会启动新线程。

## 怎么检测一个线程是否持有对象监视器

Thread 类提供了一个 holdsLock(Object obj)方法，当且仅当对象 obj 的监视器被某条线程持有 的时候才会返回 true，注意这是一个 static 方法，这意味着”某条线程”指的是当前线程。

## Runnable 和 Callable 的区别

Runnable 接口中的 run()方法的返回值是 void，它做的事情只是纯粹地去执行 run()方法中的 代码而已；Callable 接口中的 call()方法是有返回值的，是一个泛型，和 Future、FutureTask 配合可以用来获取异步执行的结果。 这其实是很有用的一个特性，因为多线程相比单线程更难、更复杂的一个重要原因就是因为 多线程充满着未知性，某条线程是否执行了？某条线程执行了多久？某条线程执行的时候我 们期望的数据是否已经赋值完毕？无法得知，我们能做的只是等待这条多线程的任务执行完 毕而已。而 Callable+Future/FutureTask 却可以方便获取多线程运行的结果，可以在等待时间 太长没获取到需要的数据的情况下取消该线程的任务。

## 什么导致线程阻塞

阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪），学过操作系统的同 学对它一定已经很熟悉了。Java 提供了大量方法来支持阻塞，下面让我们逐一分析。



## wait(),notify()和 suspend(),resume()之间的区别

初看起来它们与 suspend() 和 resume() 方法对没有什么分别，但是事实上它们是截然不同 的。区别的核心在于，前面叙述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话）， 而这一对方法则相反。上述的核心区别导致了一系列的细节上的区别。

首先，前面叙述的所有方法都隶属于 Thread 类，但是这一对却直接隶属于 Object 类，也 就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的， 因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的 wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用 任意对象的 notify()方法则导致从调 用该对象的 wait() 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真

正可执行）。

其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 synchronized 方 法或块中调用，理由也很简单，只有在 synchronized 方法或块中当前线程才占有锁，才有 锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有 锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized 方法或块中，该方法 或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但 在运行时会出现 IllegalMonitorStateException 异常。

wait() 和 notify() 方法的上述特性决定了它们经常和 synchronized 关键字一起使用，将它们 和操作系统进程间通信机制作一个比较就会发现它们的相似性：synchronized 方法或块提供 了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相

当于 block 和 wakeup 原语（这一对方法均声明为 synchronized）。它们的结合使得我们可 以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并用于解决各种复杂 的线程间通信问题。

关于 wait() 和 notify() 方法最后再说明两点： 第一：调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线 程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这 种不确定性而产生问题。

第二：除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当 然，只有获得锁的那一个线程才能进入可执行状态。

谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，suspend() 方法和不指定超时期限的 wait() 方法的调用都可能产生死锁。遗憾的是，Java 并不在语言级别上支持死锁的避免， 我们在编程中必须小心地避免死锁。

以上我们对 Java 中实现线程阻塞的各种方法作了一番分析，我们重点分析了 wait() 和 notify() 方法，因为它们的功能最强大，使用也最灵活，但是这也导致了它们的效率较低， 较容易出错。实际使用中我们应该灵活使用各种方法，以便更好地达到我们的目的。

## 产生死锁的条件

1.互斥条件：一个资源每次只能被一个进程使用。

2.请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

3.不剥夺条件:进程已获得的资源，在末使用完之前，不能强行剥夺。

4.循环等待条件:若干进程之间形成一种头尾相接的循环等待资源关系。

## 为什么 wait()方法和 notify()/notifyAll()方法要在同步块中被调用

这是 JDK 强制的，wait()方法和 notify()/notifyAll()方法在调用前都必须先获得对象的锁

## wait()方法和 notify()/notifyAll()方法在放弃对象监视器时有什么

**区别**

wait()方法和 notify()/notifyAll()方法在放弃对象监视器的时候的区别在于：wait()方法立即释 放对象监视器，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器。

## wait()与 sleep()的区别

关于这两者已经在上面进行详细的说明,这里就做个概括好了:



sleep()来自 Thread 类，和 wait()来自 Object 类。调用 sleep()方法的过程中，线程不 会释放对象锁。而 调用 wait 方法线程会释放对象锁





sleep()睡眠后不出让系统资源，wait 让其他线程可以占用 CPU





sleep(milliseconds)需要指定一个睡眠时间，时间一到会自动唤醒.而 wait()需要配合

notify()或者 notifyAll()使用







## 为什么 wait, nofity 和 nofityAll 这些方法不放在 Thread 类当中

一个很明显的原因是 JAVA 提供的锁是对象级的而不是线程级的，每个对象都有锁，通过线 程获得。如果线程需要等待某些锁那么调用对象中的 wait()方法就有意义了。如果 wait()方 法定义在 Thread 类中，线程正在等待的是哪个锁就不明显了。简单的说，由于 wait，notify 和 notifyAll 都是锁级别的操作，所以把他们定义在 Object 类中因为锁属于对象。

## 怎么唤醒一个阻塞的线程

如果线程是因为调用了 wait()、sleep()或者 join()方法而导致的阻塞，可以中断线程，并且通 过抛出 InterruptedException 来唤醒它；如果线程遇到了 IO 阻塞，无能为力，因为 IO 是操作 系统实现的，Java 代码并没有办法直接接触到操作系统。

## 什么是多线程的上下文切换

多线程的上下文切换是指 CPU 控制权由一个已经正在运行的线程切换到另外一个就绪并等 待获取 CPU 执行权的线程的过程。

## synchronized 和 ReentrantLock 的区别

synchronized 是和 if、else、for、while 一样的关键字，ReentrantLock 是类，这是二者的本质 区别。既然 ReentrantLock 是类，那么它就提供了比 synchronized 更多更灵活的特性，可以 被继承、可以有方法、可以有各种各样的类变量，ReentrantLock 比 synchronized 的扩展性体 现在几点上：

（1）ReentrantLock 可以对获取锁的等待时间进行设置，这样就避免了死锁

（2）ReentrantLock 可以获取各种锁的信息

（3）ReentrantLock 可以灵活地实现多路通知 另外，二者的锁机制其实也是不一样的:ReentrantLock 底层调用的是 Unsafe 的 park 方法加 锁，synchronized 操作的应该是对象头中 mark word。

## FutureTask 是什么

这个其实前面有提到过，FutureTask 表示一个异步运算的任务。FutureTask 里面可以传入一 个 Callable 的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经 完成、取消任务等操作。当然，由于 FutureTask 也是 Runnable 接口的实现类，所以 FutureTask 也可以放入线程池中。

## 一个线程如果出现了运行时异常怎么办?

如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：如果这个线程 持有某个某个对象的监视器，那么这个对象监视器会被立即释放。

## Java 当中有哪几种锁

自旋锁: 自旋锁在 JDK1.6 之后就默认开启了。基于之前的观察，共享数据的锁定状

态只会持续很短的时间，为了这一小段时间而去挂起和恢复线程有点浪费，所以这

里就做了一个处理，让后面请求锁的那个线程在稍等一会，但是不放弃处理器的执

行时间，看看持有锁的线程能否快速释放。为了让线程等待，所以需要让线程执行

一个忙循环也就是自旋操作。在 jdk6 之后，引入了自适应的自旋锁，也就是等待的

时间不再固定了，而是由上一次在同一个锁上的自旋时间及锁的拥有者状态来决定。

偏向锁: 在 JDK1.之后引入的一项锁优化，目的是消除数据在无竞争情况下的同步原

语。进一步提升程序的运行性能。 偏向锁就是偏心的偏，意思是这个锁会偏向第一

个获得他的线程，如果接下来的执行过程中，改锁没有被其他线程获取，则持有偏

向锁的线程将永远不需要再进行同步。偏向锁可以提高带有同步但无竞争的程序性

能，也就是说他并不一定总是对程序运行有利，如果程序中大多数的锁都是被多个

不同的线程访问，那偏向模式就是多余的，在具体问题具体分析的前提下，可以考

虑是否使用偏向锁。

轻量级锁: 为了减少获得锁和释放锁所带来的性能消耗，引入了“偏向锁”和“轻量级

锁”，所以在 Java SE1.6 里锁一共有四种状态，无锁状态，偏向锁状态，轻量级锁状

态和重量级锁状态，它会随着竞争情况逐渐升级。锁可以升级但不能降级，意味着

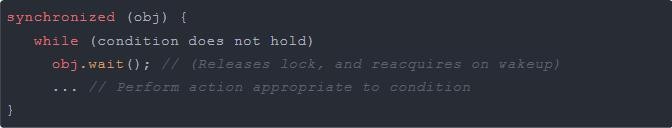
偏向锁升级成轻量级锁后不能降级成偏向锁。

## 如何在两个线程间共享数据

通过在线程之间共享对象就可以了，然后通过 wait/notify/notifyAll、await/signal/signalAll 进 行唤起和等待，比方说阻塞队列 BlockingQueue 就是为线程之间共享数据而设计的。

## 如何正确的使用 wait()？使用 if 还是 while？

wait() 方法应该在循环调用，因为当线程获取到 CPU 开始执行的时候，其他条件可能还没 有满足，所以在处理前，循环检测条件是否满足会更好。下面是一段标准的使用 wait 和 notify 方法的代码：



## 什么是线程局部变量 ThreadLocal

线程局部变量是局限于线程内部的变量，属于线程自身所有，不在多个线程间共享。Java 提供 ThreadLocal 类来支持线程局部变量，是一种实现线程安全的方式。但是在管理环境下

（如 web 服务器）使用线程局部变量的时候要特别小心，在这种情况下，工作线程的生命 周期比任何应用变量的生命周期都要长。任何线程局部变量一旦在工作完成后没有释放， Java 应用就存在内存泄露的风险。

## ThreadLoal 的作用是什么?

简 单 说 ThreadLocal 就 是 一 种 以 空 间 换 时 间 的 做 法 在 每 个 Thread 里 面 维 护 了 一 个 ThreadLocal.ThreadLocalMap 把数据进行隔离，数据不共享，自然就没有线程安全方面的问 题了。

## 生产者消费者模型的作用是什么?

（1）通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率，这是生 产者消费者模型最重要的作用。

（2）解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少， 联系越少越可以独自发展而不需要收到相互的制约。

## 写一个生产者-消费者队列

### 可以通过阻塞队列实现，也可以通过 wait-notify 来实现。

**使用阻塞队列来实现**

*//消费者*

public class Producer implements Runnable{

private final BlockingQueue<Integer> queue;

public Producer(BlockingQueue q){

this.queue=q;

}

@Override

public void run() {



try {

while (true){

Thread.sleep(1000);*//模拟耗时*

queue.put(produce());

}

}catch (InterruptedException e){

}

}

private int produce() {

int n=new Random().nextInt(10000);

System.out.println("Thread:" + Thread.currentThread().getId() produce:"



+ "



+ n);

return n;

}

}

*//消费者*

public class Consumer implements Runnable {

private final BlockingQueue<Integer> queue;

public Consumer(BlockingQueue q){

this.queue=q;

}

@Override

public void run() {

while (true){



try {

Thread.sleep(2000);*//模拟耗时*

consume(queue.take());

}catch (InterruptedException e){

}

}

}

private void consume(Integer n) {



System.out.println("Thread:" + Thread.currentThread().getId() + " consume:" + n);



}

}

*//测试*

public class Main {

public static void main(String[] args) {

BlockingQueue<Integer> queue=new ArrayBlockingQueue<Integer>(100);

Producer p=new Producer(queue);



Consumer c1=new Consumer(queue);

Consumer c2=new Consumer(queue);

Thread(p).start();



new new

new

Thread(c1).start();

Thread(c2).start();

}

}

## 使用 wait-notify 来实现

该种方式应该最经典，这里就不做说明了。

## 如果你提交任务时，线程池队列已满，这时会发生什么

如果你使用的 LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队 列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存 放任务；如果你使用的是有界队列比方说 ArrayBlockingQueue 的话，任务首先会被添加到 ArrayBlockingQueue 中 ， ArrayBlockingQueue 满 了 ， 则 会 使 用 拒 绝 策 略 RejectedExecutionHandler 处理满了的任务，默认是 AbortPolicy。

## 为什么要使用线程池

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活 地控制并发的数目。

## java 中用到的线程调度算法是什么

抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出 一个总的优先级并分配下一个时间片给某个线程执行。

## Thread.sleep(0)的作用是什么

由于 Java 采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到 CPU 控制权的 情况，为了让某些优先级比较低的线程也能获取到 CPU 控制权，可以使用 Thread.sleep(0) 手动触发一次操作系统分配时间片的操作，这也是平衡 CPU 控制权的一种操作。

## 什么是 CAS

CAS，全称为 Compare and Swap，即比较-替换。假设有三个操作数：内存值 V、旧的预期值 A、要修改的值 B，当且仅当预期值 A 和内存值 V 相同时，才会将内存值修改为 B 并返回 true， 否则什么都不做并返回 false。当然 CAS 一定要 volatile 变量配合，这样才能保证每次拿到的 变量是主内存中最新的那个值，否则旧的预期值 A 对某条线程来说，永远是一个不会变的 值 A，只要某次 CAS 操作失败，永远都不可能成功。

## 什么是乐观锁和悲观锁

乐观锁：乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作 为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的 重试逻辑。

悲观锁：悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的 锁，就像 synchronized，不管三七二十一，直接上了锁就操作资源了。

## ConcurrentHashMap 的并发度是什么?

ConcurrentHashMap 的并发度就是 segment 的大小，默认为 16，这意味着最多同时可以有 16 条线程操作 ConcurrentHashMap，这也是 ConcurrentHashMap 对 Hashtable 的最大优势， 任何情况下，Hashtable 能同时有两条线程获取 Hashtable 中的数据吗？

## ConcurrentHashMap 的工作原理

ConcurrentHashMap 在 jdk 1.6 和 jdk 1.8 实现原理是不同的。

## jdk 1.6:

ConcurrentHashMap 是线程安全的，但是与 Hashtablea 相比，实现线程安全的方式不同。 Hashtable 是通过对 hash 表结构进行锁定，是阻塞式的，当一个线程占有这个锁时，其他线 程必须阻塞等待其释放锁。ConcurrentHashMap 是采用分离锁的方式，它并没有对整个 hash 表进行锁定，而是局部锁定，也就是说当一个线程占有这个局部锁时，不影响其他线程对 hash 表其他地方的访问。

具体实现:ConcurrentHashMap 内部有一个 Segment.

## jdk 1.8

中，ConcurrentHashMap 不再使用 Segment 分离锁，而是采用一种乐观锁 CAS 算法 来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。



在 jdk 8

## CyclicBarrier 和 CountDownLatch 区别

这两个类非常类似，都在 java.util.concurrent 下，都可以用来表示代码运行到某个点上，二 者的区别在于：

CyclicBarrier 的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程 都到达了这个点，所有线程才重新运行；CountDownLatch 则不是，某线程运行到某 个点上之后，只是给某个数值-1 而已，该线程继续运行。

CyclicBarrier 只能唤起一个任务，CountDownLatch 可以唤起多个任务

CyclicBarrier 可重用，CountDownLatch 不可重用，计数值为 0 该 CountDownLatch 就 不可再用了。

## java 中的++操作符线程安全么?

不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过 程可能会出现多个线程交差。

## 你有哪些多线程开发良好的实践?

给线程命名

最小化同步范围

优先使用 volatile

尽可能使用更高层次的并发工具而非 wait 和 notify()来实现线程通信,如

BlockingQueue,Semeaphore

优先使用并发容器而非同步容器.

考虑使用线程池

# 关于 volatile 关键字

## 可以创建 Volatile 数组吗?

Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。如果 改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素， volatile 标示符就不能起到之前的保护作用了。

## volatile 能使得一个非原子操作变成原子操作吗?

一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线 程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线 程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量 的读写是原子。

一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是 64 位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然 后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变 量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如 在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会 插入一个写屏障（write barrier），读一个 volatile 变量之前，会插入一个读屏障（read barrier）。 意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之 前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新

到缓存。

## volatile 类型变量提供什么保证?

volatile 主要有两方面的作用:1.避免指令重排 2.可见性保证.例如，JVM 或者 JIT 为了获得更 好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与 其他语句重排序。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线 程是可见的。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的(低 32 位和高 32 位)，但 volatile 类型的 double 和 long 就是原子的。



# 关于集合

## Java 中的集合及其继承关系

关于集合的体系是每个人都应该烂熟于心的,尤其是对我们经常使用的 List,Map 的原理更该 如此.这里我们看这张图即可:

## 使用 wait-notify 来实现

该种方式应该最经典，这里就不做说明了。

## 如果你提交任务时，线程池队列已满，这时会发生什么

如果你使用的 LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队 列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存 放任务；如果你使用的是有界队列比方说 ArrayBlockingQueue 的话，任务首先会被添加到 ArrayBlockingQueue 中 ， ArrayBlockingQueue 满 了 ， 则 会 使 用 拒 绝 策 略 RejectedExecutionHandler 处理满了的任务，默认是 AbortPolicy。

## 为什么要使用线程池

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活 地控制并发的数目。

## java 中用到的线程调度算法是什么

抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出 一个总的优先级并分配下一个时间片给某个线程执行。

## Thread.sleep(0)的作用是什么

由于 Java 采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到 CPU 控制权的 情况，为了让某些优先级比较低的线程也能获取到 CPU 控制权，可以使用 Thread.sleep(0) 手动触发一次操作系统分配时间片的操作，这也是平衡 CPU 控制权的一种操作。

## 什么是 CAS

CAS，全称为 Compare and Swap，即比较-替换。假设有三个操作数：内存值 V、旧的预期值 A、要修改的值 B，当且仅当预期值 A 和内存值 V 相同时，才会将内存值修改为 B 并返回 true， 否则什么都不做并返回 false。当然 CAS 一定要 volatile 变量配合，这样才能保证每次拿到的 变量是主内存中最新的那个值，否则旧的预期值 A 对某条线程来说，永远是一个不会变的 值 A，只要某次 CAS 操作失败，永远都不可能成功。

## 什么是乐观锁和悲观锁

乐观锁：乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作 为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的 重试逻辑。

悲观锁：悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的 锁，就像 synchronized，不管三七二十一，直接上了锁就操作资源了。

## ConcurrentHashMap 的并发度是什么?

ConcurrentHashMap 的并发度就是 segment 的大小，默认为 16，这意味着最多同时可以有 16 条线程操作 ConcurrentHashMap，这也是 ConcurrentHashMap 对 Hashtable 的最大优势， 任何情况下，Hashtable 能同时有两条线程获取 Hashtable 中的数据吗？

## ConcurrentHashMap 的工作原理

ConcurrentHashMap 在 jdk 1.6 和 jdk 1.8 实现原理是不同的。

## jdk 1.6:

ConcurrentHashMap 是线程安全的，但是与 Hashtablea 相比，实现线程安全的方式不同。 Hashtable 是通过对 hash 表结构进行锁定，是阻塞式的，当一个线程占有这个锁时，其他线 程必须阻塞等待其释放锁。ConcurrentHashMap 是采用分离锁的方式，它并没有对整个 hash 表进行锁定，而是局部锁定，也就是说当一个线程占有这个局部锁时，不影响其他线程对 hash 表其他地方的访问。

具体实现:ConcurrentHashMap 内部有一个 Segment.

## jdk 1.8

中，ConcurrentHashMap 不再使用 Segment 分离锁，而是采用一种乐观锁 CAS 算法 来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。



在 jdk 8

## CyclicBarrier 和 CountDownLatch 区别

这两个类非常类似，都在 java.util.concurrent 下，都可以用来表示代码运行到某个点上，二 者的区别在于：



CyclicBarrier 的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程 都到达了这个点，所有线程才重新运行；CountDownLatch 则不是，某线程运行到某 个点上之后，只是给某个数值-1 而已，该线程继续运行。





CyclicBarrier 只能唤起一个任务，CountDownLatch 可以唤起多个任务





CyclicBarrier 可重用，CountDownLatch 不可重用，计数值为 0 该 CountDownLatch 就 不可再用了。



## java 中的++操作符线程安全么?

不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过 程可能会出现多个线程交差。

## 你有哪些多线程开发良好的实践?

### 1.

给线程命名

### 2.

3.

最小化同步范围

### 4.

5.

优先使用 volatile

### 6.

7.

尽可能使用更高层次的并发工具而非 wait 和 notify()来实现线程通信,如

BlockingQueue,Semeaphore

### 8.

9.

优先使用并发容器而非同步容器.

### 10.

11.

考虑使用线程池

### 12.

**关于 volatile 关键字**

**可以创建 Volatile 数组吗?**

Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。如果 改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素， volatile 标示符就不能起到之前的保护作用了。

## volatile 能使得一个非原子操作变成原子操作吗?

一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线 程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线 程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量 的读写是原子。

一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和 long 都是 64 位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然 后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变 量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如 在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会 插入一个写屏障（write barrier），读一个 volatile 变量之前，会插入一个读屏障（read barrier）。 意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之 前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新

到缓存。

## volatile 类型变量提供什么保证?

volatile 主要有两方面的作用:1.避免指令重排 2.可见性保证.例如，JVM 或者 JIT 为了获得更 好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与 其他语句重排序。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线 程是可见的。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的(低 32 位和高 32 位)，但 volatile 类型的 double 和 long 就是原子的。



# 关于集合

## Java 中的集合及其继承关系

关于集合的体系是每个人都应该烂熟于心的,尤其是对我们经常使用的 List,Map 的原理更该 如此.这里我们看这张图即可:

## 使用 wait-notify 来实现

该种方式应该最经典，这里就不做说明了。

## 如果你提交任务时，线程池队列已满，这时会发生什么

如果你使用的 LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队 列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存 放任务；如果你使用的是有界队列比方说 ArrayBlockingQueue 的话，任务首先会被添加到 ArrayBlockingQueue 中 ， ArrayBlockingQueue 满 了 ， 则 会 使 用 拒 绝 策 略 RejectedExecutionHandler 处理满了的任务，默认是 AbortPolicy。

## 为什么要使用线程池

避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活 地控制并发的数目。

## java 中用到的线程调度算法是什么

抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出 一个总的优先级并分配下一个时间片给某个线程执行。

## Thread.sleep(0)的作用是什么

由于 Java 采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到 CPU 控制权的 情况，为了让某些优先级比较低的线程也能获取到 CPU 控制权，可以使用 Thread.sleep(0) 手动触发一次操作系统分配时间片的操作，这也是平衡 CPU 控制权的一种操作。

## 什么是 CAS

CAS，全称为 Compare and Swap，即比较-替换。假设有三个操作数：内存值 V、旧的预期值 A、要修改的值 B，当且仅当预期值 A 和内存值 V 相同时，才会将内存值修改为 B 并返回 true， 否则什么都不做并返回 false。当然 CAS 一定要 volatile 变量配合，这样才能保证每次拿到的

变量是主内存中最新的那个值，否则旧的预期值 A 对某条线程来说，永远是一个不会变的 值 A，只要某次 CAS 操作失败，永远都不可能成功。

## 什么是乐观锁和悲观锁

乐观锁：乐观锁认为竞争不总是会发生，因此它不需要持有锁，将比较-替换这两个动作作 为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的 重试逻辑。

悲观锁：悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的 锁，就像 synchronized，不管三七二十一，直接上了锁就操作资源了。

## ConcurrentHashMap 的并发度是什么?

ConcurrentHashMap 的并发度就是 segment 的大小，默认为 16，这意味着最多同时可以有 16 条线程操作 ConcurrentHashMap，这也是 ConcurrentHashMap 对 Hashtable 的最大优势， 任何情况下，Hashtable 能同时有两条线程获取 Hashtable 中的数据吗？

## ConcurrentHashMap 的工作原理

ConcurrentHashMap 在 jdk 1.6 和 jdk 1.8 实现原理是不同的。

## jdk 1.6:

ConcurrentHashMap 是线程安全的，但是与 Hashtablea 相比，实现线程安全的方式不同。 Hashtable 是通过对 hash 表结构进行锁定，是阻塞式的，当一个线程占有这个锁时，其他线 程必须阻塞等待其释放锁。ConcurrentHashMap 是采用分离锁的方式，它并没有对整个 hash 表进行锁定，而是局部锁定，也就是说当一个线程占有这个局部锁时，不影响其他线程对 hash 表其他地方的访问。

具体实现:ConcurrentHashMap 内部有一个 Segment.

## jdk 1.8

中，ConcurrentHashMap 不再使用 Segment 分离锁，而是采用一种乐观锁 CAS 算法 来实现同步问题，但其底层还是“数组+链表->红黑树”的实现。



在 jdk 8

## CyclicBarrier 和 CountDownLatch 区别

这两个类非常类似，都在 java.util.concurrent 下，都可以用来表示代码运行到某个点上，二 者的区别在于：



CyclicBarrier 的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程 都到达了这个点，所有线程才重新运行；CountDownLatch 则不是，某线程运行到某 个点上之后，只是给某个数值-1 而已，该线程继续运行。





CyclicBarrier 只能唤起一个任务，CountDownLatch 可以唤起多个任务





CyclicBarrier 可重用，CountDownLatch 不可重用，计数值为 0 该 CountDownLatch 就 不可再用了。



## java 中的++操作符线程安全么?

不是线程安全的操作。它涉及到多个指令，如读取变量值，增加，然后存储回内存，这个过 程可能会出现多个线程交差。

## 你有哪些多线程开发良好的实践?

### 1.

给线程命名

### 2.

3.

最小化同步范围

### 4.

5.

优先使用 volatile

### 6.

7.

尽可能使用更高层次的并发工具而非 wait 和 notify()来实现线程通信,如

BlockingQueue,Semeaphore

### 8.

9.

优先使用并发容器而非同步容器.

### 10.

11.

考虑使用线程池

### 12.

**关于 volatile 关键字**

**可以创建 Volatile 数组吗?**

Java 中可以创建 volatile 类型数组，不过只是一个指向数组的引用，而不是整个数组。如果 改变引用指向的数组，将会受到 volatile 的保护，但是如果多个线程同时改变数组的元素， volatile 标示符就不能起到之前的保护作用了。

## volatile 能使得一个非原子操作变成原子操作吗?

一个典型的例子是在类中有一个 long 类型的成员变量。如果你知道该成员变量会被多个线 程访问，如计数器、价格等，你最好是将其设置为 volatile。为什么？因为 Java 中读取 long 类型变量不是原子的，需要分成两步，如果一个线程正在修改该 long 变量的值，另一个线 程可能只能看到该值的一半（前 32 位）。但是对一个 volatile 型的 long 或 double 变量 的读写是原子。

一种实践是用 volatile 修饰 long 和 double 变量，使其能按原子类型来读写。double 和

long 都是 64 位宽，因此对这两种类型的读是分为两部分的，第一次读取第一个 32 位，然

后再读剩下的 32 位，这个过程不是原子的，但 Java 中 volatile 型的 long 或 double 变 量的读写是原子的。volatile 修复符的另一个作用是提供内存屏障（memory barrier），例如 在分布式框架中的应用。简单的说，就是当你写一个 volatile 变量之前，Java 内存模型会 插入一个写屏障（write barrier），读一个 volatile 变量之前，会插入一个读屏障（read barrier）。 意思就是说，在你写一个 volatile 域时，能保证任何线程都能看到你写的值，同时，在写之 前，也能保证任何数值的更新对所有线程是可见的，因为内存屏障会将其他所有写的值更新

到缓存。

## volatile 类型变量提供什么保证?

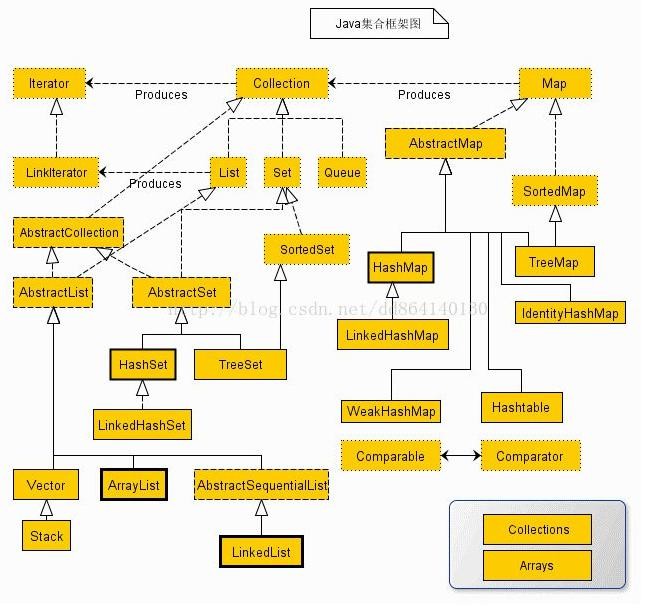
volatile 主要有两方面的作用:1.避免指令重排 2.可见性保证.例如，JVM 或者 JIT 为了获得更 好的性能会对语句重排序，但是 volatile 类型变量即使在没有同步块的情况下赋值也不会与 其他语句重排序。 volatile 提供 happens-before 的保证，确保一个线程的修改能对其他线 程是可见的。某些情况下，volatile 还能提供原子性，如读 64 位数据类型，像 long 和 double 都不是原子的(低 32 位和高 32 位)，但 volatile 类型的 double 和 long 就是原子的。



# 关于集合

## Java 中的集合及其继承关系

关于集合的体系是每个人都应该烂熟于心的,尤其是对我们经常使用的 List,Map 的原理更该 如此.这里我们看这张图即可:



## poll()方法和 remove()方法区别？

poll() 和 remove() 都是从队列中取出一个元素，但是 poll() 在获取元素失败的时候会返回 空，但是 remove() 失败的时候会抛出异常。

## LinkedHashMap 和 PriorityQueue 的区别

PriorityQueue 是一个优先级队列,保证最高或者最低优先级的的元素总是在队列头部，但是 LinkedHashMap 维持的顺序是元素插入的顺序。当遍历一个 PriorityQueue 时，没有任何顺 序保证，但是 LinkedHashMap 课保证遍历顺序是元素插入的顺序。

## WeakHashMap 与 HashMap 的区别是什么?

WeakHashMap 的工作与正常的 HashMap 类似，但是使用弱引用作为 key，意思就是当 key

对象没有任何引用时，key/value 将会被回收。

## ArrayList 和 LinkedList 的区别?

最明显的区别是 ArrrayList 底层的数据结构是数组，支持随机访问，而 LinkedList 的底层数 据结构是双向循环链表，不支持随机访问。使用下标访问一个元素，ArrayList 的时间复杂 度是 O(1)，而 LinkedList 是 O(n)。

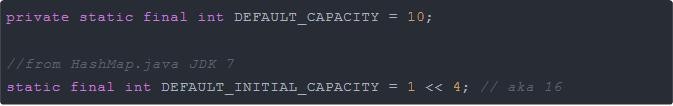
## ArrayList 和 Array 有什么区别?

Array 可以容纳基本类型和对象，而 ArrayList 只能容纳对象。

Array 是指定大小的，而 ArrayList 大小是固定的

## ArrayList 和 HashMap 默认大小?

在 Java 7 中，ArrayList 的默认大小是 10 个元素，HashMap 的默认大小是 16 个元素（必 须是 2 的幂）。这就是 Java 7 中 ArrayList 和 HashMap 类的代码片段。



## Comparator 和 Comparable 的区别?

Comparable 接口用于定义对象的自然顺序，而 comparator 通常用于定义用户定制的顺序。 Comparable 总是只有一个，但是可以有多个 comparator 来定义对象的顺序。

## 如何实现集合排序?

你可以使用有序集合，如 TreeSet 或 TreeMap，你也可以使用有顺序的的集合，如 list，然 后通过 Collections.sort() 来排序。

## 如何打印数组内容

你可以使用 Arrays.toString() 和 Arrays.deepToString() 方法来打印数组。由于数组没有实现 toString() 方法，所以如果将数组传递给 System.out.println() 方法，将无法打印出数组的内 容，但是 Arrays.toString() 可以打印每个元素。

## LinkedList 的是单向链表还是双向?

双向循环列表，具体实现自行查阅源码。

## TreeMap 是实现原理

### 采用红黑树实现，具体实现自行查阅源码。

**遍历 ArrayList 时如何正确移除一个元素**

该问题的关键在于面试者使用的是 ArrayList 的 remove() 还是 Iterator 的 remove()方法。 这有一段示例代码，是使用正确的方式来实现在遍历的过程中移除元素，而不会出现 ConcurrentModificationException 异常的示例代码。

## 什么是 ArrayMap?它和 HashMap 有什么区别?

ArrayMap 是 Android SDK 中提供的，非 Android 开发者可以略过。 ArrayMap 是用两个数组来模拟 map，更少的内存占用空间,更高的效率。

## HashMap 的实现原理

1. HashMap 概述： HashMap 是基于哈希表的 Map 接口的非同步实现。此实现提供所有可 选的映射操作，并允许使用 null 值和 null 键。此类不保证映射的顺序，特别是它不保证该顺 序恒久不变。 2. HashMap 的数据结构： 在 java 编程语言中，最基本的结构就是两种，一个是数组，另外

一个是模拟指针（引用），所有的数据结构都可以用这两个基本结构来构造的，HashMap 也不例外。HashMap 实际上是一个“链表散列”的数据结构，即数组和链表的结合体。

当我们往 Hashmap 中 put 元素时,首先根据 key 的 hashcode 重新计算 hash 值,根绝 hash 值得 到这个元素在数组中的位置(下标),如果该数组在该位置上已经存放了其他元素,那么在这个 位置上的元素将以链表的形式存放,新加入的放在链头,最先加入的放入链尾.如果数组中该 位置没有元素,就直接将该元素放到数组的该位置上.



需要注意 Jdk 1.8 中对 HashMap 的实现做了优化,当链表中的节点数据超过八个之后,该链表 会转为红黑树来提高查询效率,从原来的 O(n)到 O(logn)

## 你了解 Fail-Fast 机制吗？

Fail-Fast 即我们常说的快速失败，

## Fail-fast 和 Fail-safe 有什么区别

Iterator 的 fail-fast 属性与当前的集合共同起作用，因此它不会受到集合中任何改动的影响。 Java.util 包中的所有集合类都被设计为 fail->fast 的，而 java.util.concurrent 中的集合类都为 fail-safe 的 。 当 检 测 到 正 在 遍 历 的 集 合 的 结 构 被 改 变 时 ， Fail-fast 迭 代 器 抛 出 ConcurrentModificationException ， 而 fail-safe 迭 代 器 从 不 抛 出 ConcurrentModificationException。

# 关于日期

## SimpleDateFormat 是线程安全的吗?

非常不幸，DateFormat 的所有实现，包括 SimpleDateFormat 都不是线程安全的，因此你不 应该在多线程序中使用，除非是在对外线程安全的环境中使用，如 将 SimpleDateFormat 限 制在 ThreadLocal 中。如果你不这么做，在解析或者格式化日期的时候，可能会获取到一个 不正确的结果。因此，从日期、时间处理的所有实践来说，我强力推荐 joda-time 库。



## 如何格式化日期?

Java 中，可以使用 SimpleDateFormat 类或者 joda-time 库来格式日期。DateFormat 类允 许你使用多种流行的格式来格式化日期。参见答案中的示例代码，代码中演示了将日期格式 化成不同的格式，如 dd-MM-yyyy 或 ddMMyyyy。

# 关于异常

## 简单描述 java 异常体系

相比没有人不了解异常体系,关于异常体系的更多信息可以见

## 什么是异常链

详情直接参见上面的白话异常机制，不做解释了。

## throw 和 throws 的区别

throw 用于主动抛出 java.lang.Throwable 类的一个实例化对象，意思是说你可以通过关键字

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| throw | 抛 | 出 | 一 | 个 | Error | 或 | 者 | 一 | 个 | Exception | ， | 如 | ： | throw | new |

,

IllegalArgumentException(“size must be multiple of 2″)

而 throws 的作用是作为方法声明和签名的一部分，方法被抛出相应的异常以便调用者能处 理。Java 中，任何未处理的受检查异常强制在 throws 子句中声明。

# 关于序列化

## Java 中，Serializable 与 Externalizable 的区别

Serializable 接口是一个序列化 Java 类的接口，以便于它们可以在网络上传输或者可以将它 们的状态保存在磁盘上，是 JVM 内嵌的默认序列化方式，成本高、脆弱而且不安全。 Externalizable 允许你控制整个序列化过程，指定特定的二进制格式，增加安全机制。

# 关于 JVM

**其他**

## java 当中采用的是大端还是小端?

**XML 解析的几种方式和特点**

DOM, SAX, PULL 三种解析方式：

DOM:消耗内存：先把 xml 文档都读到内存中，然后再用 DOM API 来访问树形结构， 并获取数据。这个写起来很简单，但是很消耗内存。要是数据过大，手机不够牛逼， 可能手机直接死机

SAX:解析效率高，占用内存少，基于事件驱动的：更加简单地说就是对文档进行顺 序扫描，当扫描到文档(document)开始与结束、元素(element)开始与结束、文档

(document)结束等地方时通知事件处理函数，由事件处理函数做相应动作，然后继

续同样的扫描，直至文档结束。

PULL:与 SAX 类似，也是基于事件驱动，我们可以调用它的 next（）方法，来获取 下一个解析事件（就是开始文档，结束文档，开始标签，结束标签），当处于某个 元素时可以调用 XmlPullParser 的 getAttributte()方法来获取属性的值，也可调用它的 nextText()获取本节点的值。



## Maven 和 ANT 有什么区别?

虽然两者都是构建工具，都用于创建 Java 应用，但是 Maven 做的事情更多，在基于“约定 优于配置”的概念下，提供标准的 Java 项目结构，同时能为应用自动管理依赖（应用中所依 赖的 JAR 文件。

## JDBC 最佳实践

优先使用批量操作来插入和更新数据

使用 PreparedStatement 来避免 SQL 漏洞

使用数据连接池

通过列名来获取结果集

## IO 操作最佳实践

使用有缓冲的 IO 类,不要单独读取字节或字符

使用 NIO 和 NIO 2 或者 AIO,而非 BIO

在 finally 中关闭流

使用内存映射文件获取更快的 IO