# Java面试题全集（上）

## ****解释内存中的栈(stack)、堆(heap)和方法区(method area)的用法。****

答：通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用JVM中的栈空间；而通过new关键字和构造器创建的对象则放在堆空间，堆是垃圾收集器管理的主要区域，由于现在的垃圾收集器都采用分代收集算法，所以堆空间还可以细分为新生代和老生代，再具体一点可以分为Eden、Survivor（又可分为From Survivor和To Survivor）、Tenured；方法区和堆都是各个线程共享的内存区域，用于存储已经被JVM加载的类信息、常量、静态变量、JIT编译器编译后的代码等数据；程序中的字面量（literal）如直接书写的100、"hello"和常量都是放在常量池中，常量池是方法区的一部分，。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，栈和堆的大小都可以通过JVM的启动参数来进行调整，栈空间用光了会引发StackOverflowError，而堆和常量池空间不足则会引发OutOfMemoryError。

String str = new String("hello");

上面的语句中变量str放在栈上，用new创建出来的字符串对象放在堆上，而"hello"这个字面量是放在方法区的。

**补充1：**较新版本的Java（从Java 6的某个更新开始）中，由于JIT编译器的发展和"逃逸分析"技术的逐渐成熟，栈上分配、标量替换等优化技术使得对象一定分配在堆上这件事情已经变得不那么绝对了。

**补充2**：运行时常量池相当于Class文件常量池具有动态性，Java语言并不要求常量一定只有编译期间才能产生，运行期间也可以将新的常量放入池中，String类的intern()方法就是这样的。

看看下面代码的执行结果是什么并且比较一下Java 7以前和以后的运行结果是否一致。

String s1 = new StringBuilder("go").append("od").toString();

System.out.println(s1.intern() == s1);

String s2 = new StringBuilder("ja").append("va").toString();

System.out.println(s2.intern() == s2);

## **Math.round(11.5) 等于多少？Math.round(-11.5)等于多少？**

答：Math.round(11.5)的返回值是12，Math.round(-11.5)的返回值是-11。四舍五入的原理是在参数上加0.5然后进行下取整。

## **两个对象值相同(x.equals(y) == true)，但却可有不同的hash code，这句话对不对？**

答：不对，如果两个对象x和y满足x.equals(y) == true，它们的哈希码（hash code）应当相同。Java对于eqauls方法和hashCode方法是这样规定的：(1)如果两个对象相同（equals方法返回true），那么它们的hashCode值一定要相同；(2)如果两个对象的hashCode相同，它们并不一定相同。当然，你未必要按照要求去做，但是如果你违背了上述原则就会发现在使用容器时，相同的对象可以出现在Set集合中，同时增加新元素的效率会大大下降（对于使用哈希存储的系统，如果哈希码频繁的冲突将会造成存取性能急剧下降）。

**补充：**关于equals和hashCode方法，很多Java程序都知道，但很多人也就是仅仅知道而已，在Joshua Bloch的大作《Effective Java》（很多软件公司，《Effective Java》、《Java编程思想》以及《重构：改善既有代码质量》是Java程序员必看书籍，如果你还没看过，那就赶紧去[亚马逊](http://z.cn/)买一本吧）中是这样介绍equals方法的：  
首先equals方法必须满足自反性（x.equals(x)必须返回true）、对称性（x.equals(y)返回true时，y.equals(x)也必须返回true）、传递性（x.equals(y)和y.equals(z)都返回true时，x.equals(z)也必须返回true）和一致性（当x和y引用的对象信息没有被修改时，多次调用x.equals(y)应该得到同样的返回值），而且对于任何非null值的引用x，x.equals(null)必须返回false。  
实现高质量的equals方法的诀窍包括：  
1. 使用==操作符检查"参数是否为这个对象的引用"；  
2. 使用instanceof操作符检查"参数是否为正确的类型"；  
3. 对于类中的关键属性，检查参数传入对象的属性是否与之相匹配；  
4. 编写完equals方法后，问自己它是否满足对称性、传递性、一致性；  
5. 重写equals时总是要重写hashCode；  
6. 不要将equals方法参数中的Object对象替换为其他的类型，在重写时不要忘掉@Override注解。

## **当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递？**

答：是值传递。Java语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的属性可以在被调用过程中被改变，但对对象引用的改变是不会影响到调用者的。C++和C#中可以通过传引用或传输出参数来改变传入的参数的值。在C#中可以编写如下所示的代码，但是在Java中却做不到。

using System;

namespace CS01 {

class Program {

public static void swap(ref int x, ref int y) {

int temp = x;

x = y;

y = temp;

}

public static void Main (string[] args) {

int a = 5, b = 10;

swap (ref a, ref b);

// a = 10, b = 5;

Console.WriteLine ("a = {0}, b = {1}", a, b);

}

}

}

说明：Java中没有传引用实在是非常的不方便，这一点在Java 8中仍然没有得到改进，正是如此在Java编写的代码中才会出现大量的Wrapper类（将需要通过方法调用修改的引用置于一个Wrapper类中，再将Wrapper对象传入方法），这样的做法只会让代码变得臃肿，尤其是让从C和C++转型为Java程序员的开发者无法容忍。

## **Java 中会存在内存泄漏吗，请简单描述。**

答：理论上Java因为有垃圾回收机制（GC）不会存在内存泄露问题（这也是Java被广泛使用于服务器端编程的一个重要原因）；然而在实际开发中，可能会存在无用但可达的对象，这些对象不能被GC回收，因此也会导致内存泄露的发生。例如Hibernate的Session（一级缓存）中的对象属于持久态，垃圾回收器是不会回收这些对象的，然而这些对象中可能存在无用的垃圾对象，如果不及时关闭（close）或清空（flush）一级缓存就可能导致内存泄露。下面例子中的代码也会导致内存泄露。

import java.util.Arrays;

import java.util.EmptyStackException;

public class MyStack<T> {

private T[] elements;

private int size = 0;

private static final int INIT\_CAPACITY = 16;

public MyStack() {

elements = (T[]) new Object[INIT\_CAPACITY];

}

public void push(T elem) {

ensureCapacity();

elements[size++] = elem;

}

public T pop() {

if(size == 0)

throw new EmptyStackException();

return elements[--size];

}

private void ensureCapacity() {

if(elements.length == size) {

elements = Arrays.copyOf(elements, 2 \* size + 1);

}

}

}

上面的代码实现了一个栈（先进后出（FILO））结构，乍看之下似乎没有什么明显的问题，它甚至可以通过你编写的各种单元测试。然而其中的pop方法却存在内存泄露的问题，当我们用pop方法弹出栈中的对象时，该对象不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，因为栈内部维护着对这些对象的过期引用（obsolete reference）。在支持垃圾回收的语言中，内存泄露是很隐蔽的，这种内存泄露其实就是无意识的对象保持。如果一个对象引用被无意识的保留起来了，那么垃圾回收器不会处理这个对象，也不会处理该对象引用的其他对象，即使这样的对象只有少数几个，也可能会导致很多的对象被排除在垃圾回收之外，从而对性能造成重大影响，极端情况下会引发Disk Paging（物理内存与硬盘的虚拟内存交换数据），甚至造成OutOfMemoryError。

## **如何实现对象克隆？**

答：有两种方式：   
  1). 实现Cloneable接口并重写Object类中的clone()方法；   
  2). 实现Serializable接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆，代码如下。

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

public class MyUtil {

private MyUtil() {

throw new AssertionError();

}

@SuppressWarnings("unchecked")

public static <T extends Serializable> T clone(T obj) throws Exception {

ByteArrayOutputStream bout = new ByteArrayOutputStream();

ObjectOutputStream oos = new ObjectOutputStream(bout);

oos.writeObject(obj);

ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());

ObjectInputStream ois = new ObjectInputStream(bin);

return (T) ois.readObject();

// 说明：调用ByteArrayInputStream或ByteArrayOutputStream对象的close方法没有任何意义

// 这两个基于内存的流只要垃圾回收器清理对象就能够释放资源，这一点不同于对外部资源（如文件流）的释放

}

}

下面是测试代码：

import java.io.Serializable;

/\*\*

\* 人类

\* @author 骆昊

\*

\*/

class Person implements Serializable {

private static final long serialVersionUID = -9102017020286042305L;

private String name; // 姓名

private int age; // 年龄

private Car car; // 座驾

public Person(String name, int age, Car car) {

this.name = name;

this.age = age;

this.car = car;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public Car getCar() {

return car;

}

public void setCar(Car car) {

this.car = car;

}

@Override

public String toString() {

return "Person [name=" + name + ", age=" + age + ", car=" + car + "]";

}

}

/\*\*

\* 小汽车类

\* @author 骆昊

\*

\*/

class Car implements Serializable {

private static final long serialVersionUID = -5713945027627603702L;

private String brand; // 品牌

private int maxSpeed; // 最高时速

public Car(String brand, int maxSpeed) {

this.brand = brand;

this.maxSpeed = maxSpeed;

}

public String getBrand() {

return brand;

}

public void setBrand(String brand) {

this.brand = brand;

}

public int getMaxSpeed() {

return maxSpeed;

}

public void setMaxSpeed(int maxSpeed) {

this.maxSpeed = maxSpeed;

}

@Override

public String toString() {

return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]";

}

}

class CloneTest {

public static void main(String[] args) {

try {

Person p1 = new Person("Hao LUO", 33, new Car("Benz", 300));

Person p2 = MyUtil.clone(p1); // 深度克隆

p2.getCar().setBrand("BYD");

// 修改克隆的Person对象p2关联的汽车对象的品牌属性

// 原来的Person对象p1关联的汽车不会受到任何影响

// 因为在克隆Person对象时其关联的汽车对象也被克隆了

System.out.println(p1);

} catch (Exception e) {

e.printStackTrace();

}

}

}

**注意：**基于序列化和反序列化实现的克隆不仅仅是深度克隆，更重要的是通过泛型限定，可以检查出要克隆的对象是否支持序列化，这项检查是编译器完成的，不是在运行时抛出异常，这种是方案明显优于使用Object类的clone方法克隆对象。让问题在编译的时候暴露出来总是好过把问题留到运行时。

## GC是什么？为什么要有GC？

答：GC是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java语言没有提供释放已分配内存的显示操作方法。Java程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc() 或Runtime.getRuntime().gc() ，但JVM可以屏蔽掉显示的垃圾回收调用。   
垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。在Java诞生初期，垃圾回收是Java最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今Java的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得iOS的系统比Android系统有更好的用户体验，其中一个深层次的原因就在于Android系统中垃圾回收的不可预知性。

**补充：**垃圾回收机制有很多种，包括：分代复制垃圾回收、标记垃圾回收、增量垃圾回收等方式。标准的Java进程既有栈又有堆。栈保存了原始型局部变量，堆保存了要创建的对象。Java平台对堆内存回收和再利用的基本算法被称为标记和清除，但是Java对其进行了改进，采用“分代式垃圾收集”。这种方法会跟Java对象的生命周期将堆内存划分为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：   
- 伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。   
- 幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。   
- 终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。

与垃圾回收相关的JVM参数：

* -Xms / -Xmx — 堆的初始大小 / 堆的最大大小
* -Xmn — 堆中年轻代的大小
* -XX:-DisableExplicitGC — 让System.gc()不产生任何作用
* -XX:+PrintGCDetails — 打印GC的细节
* -XX:+PrintGCDateStamps — 打印GC操作的时间戳
* -XX:NewSize / XX:MaxNewSize — 设置新生代大小/新生代最大大小
* -XX:NewRatio — 可以设置老生代和新生代的比例
* -XX:PrintTenuringDistribution — 设置每次新生代GC后输出幸存者乐园中对象年龄的分布
* -XX:InitialTenuringThreshold / -XX:MaxTenuringThreshold：设置老年代阀值的初始值和最大值
* -XX:TargetSurvivorRatio：设置幸存区的目标使用率

## **日期和时间：** **- 如何取得年月日、小时分钟秒？**  **- 如何取得从1970年1月1日0时0分0秒到现在的毫秒数？**  **- 如何取得某月的最后一天？**  **- 如何格式化日期？**

答：   
问题1：创建java.util.Calendar 实例，调用其get()方法传入不同的参数即可获得参数所对应的值。Java 8中可以使用java.time.LocalDateTimel来获取，代码如下所示。

public class DateTimeTest {

public static void main(String[] args) {

Calendar cal = Calendar.getInstance();

System.out.println(cal.get(Calendar.YEAR));

System.out.println(cal.get(Calendar.MONTH)); // 0 - 11

System.out.println(cal.get(Calendar.DATE));

System.out.println(cal.get(Calendar.HOUR\_OF\_DAY));

System.out.println(cal.get(Calendar.MINUTE));

System.out.println(cal.get(Calendar.SECOND));

// Java 8

LocalDateTime dt = LocalDateTime.now();

System.out.println(dt.getYear());

System.out.println(dt.getMonthValue()); // 1 - 12

System.out.println(dt.getDayOfMonth());

System.out.println(dt.getHour());

System.out.println(dt.getMinute());

System.out.println(dt.getSecond());

}

}

问题2：以下方法均可获得该毫秒数。

Calendar.getInstance().getTimeInMillis();

System.currentTimeMillis();

Clock.systemDefaultZone().millis(); // Java 8

问题3：代码如下所示。

Calendar time = Calendar.getInstance();

time.getActualMaximum(Calendar.DAY\_OF\_MONTH);

问题4：利用java.text.DataFormat 的子类（如SimpleDateFormat类）中的format(Date)方法可将日期格式化。Java 8中可以用java.time.format.DateTimeFormatter来格式化时间日期，代码如下所示。

import java.text.SimpleDateFormat;

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

import java.util.Date;

class DateFormatTest {

public static void main(String[] args) {

SimpleDateFormat oldFormatter = new SimpleDateFormat("yyyy/MM/dd");

Date date1 = new Date();

System.out.println(oldFormatter.format(date1));

// Java 8

DateTimeFormatter newFormatter = DateTimeFormatter.ofPattern("yyyy/MM/dd");

LocalDate date2 = LocalDate.now();

System.out.println(date2.format(newFormatter));

}

}

补充：Java的时间日期API一直以来都是被诟病的东西，为了解决这一问题，Java 8中引入了新的时间日期API，其中包括LocalDate、LocalTime、LocalDateTime、Clock、Instant等类，这些的类的设计都使用了不变模式，因此是线程安全的设计。如果不理解这些内容，可以参考我的另一篇文章[《关于Java并发编程的总结和思考》](http://blog.csdn.net/jackfrued/article/details/44499227)。

## **打印昨天的当前时刻。**

答：

import java.util.Calendar;

class YesterdayCurrent {

public static void main(String[] args){

Calendar cal = Calendar.getInstance();

cal.add(Calendar.DATE, -1);

System.out.println(cal.getTime());

}

}

在Java 8中，可以用下面的代码实现相同的功能。

import java.time.LocalDateTime;

class YesterdayCurrent {

public static void main(String[] args) {

LocalDateTime today = LocalDateTime.now();

LocalDateTime yesterday = today.minusDays(1);

System.out.println(yesterday);

}

}

## **List、Set、Map是否继承自Collection接口？**

答：List、Set 是，Map 不是。Map是键值对映射容器，与List和Set有明显的区别，而Set存储的零散的元素且不允许有重复元素（数学中的集合也是如此），List是线性结构的容器，适用于按数值索引访问元素的情形。

## **阐述ArrayList、Vector、LinkedList的存储性能和特性。**

答：ArrayList 和Vector都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引数据快而插入数据慢，Vector中的方法由于添加了synchronized修饰，因此Vector是线程安全的容器，但性能上较ArrayList差，因此已经是Java中的遗留容器。LinkedList使用双向链表实现存储（将内存中零散的内存单元通过附加的引用关联起来，形成一个可以按序号索引的线性结构，这种链式存储方式与数组的连续存储方式相比，内存的利用率更高），按序号索引数据需要进行前向或后向遍历，但是插入数据时只需要记录本项的前后项即可，所以插入速度较快。Vector属于遗留容器（Java早期的版本中提供的容器，除此之外，Hashtable、Dictionary、BitSet、Stack、Properties都是遗留容器），已经不推荐使用，但是由于ArrayList和LinkedListed都是非线程安全的，如果遇到多个线程操作同一个容器的场景，则可以通过工具类Collections中的synchronizedList方法将其转换成线程安全的容器后再使用（这是对装潢模式的应用，将已有对象传入另一个类的构造器中创建新的对象来增强实现）。

**补充：**遗留容器中的Properties类和Stack类在设计上有严重的问题，Properties是一个键和值都是字符串的特殊的键值对映射，在设计上应该是关联一个Hashtable并将其两个泛型参数设置为String类型，但是Java API中的Properties直接继承了Hashtable，这很明显是对继承的滥用。这里复用代码的方式应该是Has-A关系而不是Is-A关系，另一方面容器都属于工具类，继承工具类本身就是一个错误的做法，使用工具类最好的方式是Has-A关系（关联）或Use-A关系（依赖）。同理，Stack类继承Vector也是不正确的。Sun公司的工程师们也会犯这种低级错误，让人唏嘘不已。

## **Collection和Collections的区别？**

答：Collection是一个接口，它是Set、List等容器的父接口；Collections是个一个工具类，提供了一系列的静态方法来辅助容器操作，这些方法包括对容器的搜索、排序、线程安全化等等。

## **List、Map、Set三个接口存取元素时，各有什么特点？**

答：List以特定索引来存取元素，可以有重复元素。Set不能存放重复元素（用对象的equals()方法来区分元素是否重复）。Map保存键值对（key-value pair）映射，映射关系可以是一对一或多对一。Set和Map容器都有基于哈希存储和排序树的两种实现版本，基于哈希存储的版本理论存取时间复杂度为O(1)，而基于排序树版本的实现在插入或删除元素时会按照元素或元素的键（key）构成排序树从而达到排序和去重的效果。

## **TreeMap和TreeSet在排序时如何比较元素？Collections工具类中的sort()方法如何比较元素？**

答：TreeSet要求存放的对象所属的类必须实现Comparable接口，该接口提供了比较元素的compareTo()方法，当插入元素时会回调该方法比较元素的大小。TreeMap要求存放的键值对映射的键必须实现Comparable接口从而根据键对元素进行排序。Collections工具类的sort方法有两种重载的形式，第一种要求传入的待排序容器中存放的对象比较实现Comparable接口以实现元素的比较；第二种不强制性的要求容器中的元素必须可比较，但是要求传入第二个参数，参数是Comparator接口的子类型（需要重写compare方法实现元素的比较），相当于一个临时定义的排序规则，其实就是通过接口注入比较元素大小的算法，也是对回调模式的应用（Java中对函数式编程的支持）。   
例子1：

public class Student implements Comparable<Student> {

private String name; // 姓名

private int age; // 年龄

public Student(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return "Student [name=" + name + ", age=" + age + "]";

}

@Override

public int compareTo(Student o) {

return this.age - o.age; // 比较年龄(年龄的升序)

}

}

import java.util.Set;

import java.util.TreeSet;

class Test01 {

public static void main(String[] args) {

Set<Student> set = new TreeSet<>(); // Java 7的钻石语法(构造器后面的尖括号中不需要写类型)

set.add(new Student("Hao LUO", 33));

set.add(new Student("XJ WANG", 32));

set.add(new Student("Bruce LEE", 60));

set.add(new Student("Bob YANG", 22));

for(Student stu : set) {

System.out.println(stu);

}

// 输出结果:

// Student [name=Bob YANG, age=22]

// Student [name=XJ WANG, age=32]

// Student [name=Hao LUO, age=33]

// Student [name=Bruce LEE, age=60]

}

}

例子2：

public class Student {

private String name; // 姓名

private int age; // 年龄

public Student(String name, int age) {

this.name = name;

this.age = age;

}

/\*\*

\* 获取学生姓名

\*/

public String getName() {

return name;

}

/\*\*

\* 获取学生年龄

\*/

public int getAge() {

return age;

}

@Override

public String toString() {

return "Student [name=" + name + ", age=" + age + "]";

}

}

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

class Test02 {

public static void main(String[] args) {

List<Student> list = new ArrayList<>(); // Java 7的钻石语法(构造器后面的尖括号中不需要写类型)

list.add(new Student("Hao LUO", 33));

list.add(new Student("XJ WANG", 32));

list.add(new Student("Bruce LEE", 60));

list.add(new Student("Bob YANG", 22));

// 通过sort方法的第二个参数传入一个Comparator接口对象

// 相当于是传入一个比较对象大小的算法到sort方法中

// 由于Java中没有函数指针、仿函数、委托这样的概念

// 因此要将一个算法传入一个方法中唯一的选择就是通过接口回调

Collections.sort(list, new Comparator<Student> () {

@Override

public int compare(Student o1, Student o2) {

return o1.getName().compareTo(o2.getName()); // 比较学生姓名

}

});

for(Student stu : list) {

System.out.println(stu);

}

// 输出结果:

// Student [name=Bob YANG, age=22]

// Student [name=Bruce LEE, age=60]

// Student [name=Hao LUO, age=33]

// Student [name=XJ WANG, age=32]

}

}

## **Thread类的sleep()方法和对象的wait()方法都可以让线程暂停执行，它们有什么区别?**

答：sleep()方法（休眠）是线程类（Thread）的静态方法，调用此方法会让当前线程暂停执行指定的时间，将执行机会（CPU）让给其他线程，但是对象的锁依然保持，因此休眠时间结束后会自动恢复（线程回到就绪状态，请参考第66题中的线程状态转换图）。wait()是Object类的方法，调用对象的wait()方法导致当前线程放弃对象的锁（线程暂停执行），进入对象的等待池（wait pool），只有调用对象的notify()方法（或notifyAll()方法）时才能唤醒等待池中的线程进入等锁池（lock pool），如果线程重新获得对象的锁就可以进入就绪状态。

**补充：**可能不少人对什么是进程，什么是线程还比较模糊，对于为什么需要多线程编程也不是特别理解。简单的说：进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，是操作系统进行资源分配和调度的一个独立单位；线程是进程的一个实体，是CPU调度和分派的基本单位，是比进程更小的能独立运行的基本单位。线程的划分尺度小于进程，这使得多线程程序的并发性高；进程在执行时通常拥有独立的内存单元，而线程之间可以共享内存。使用多线程的编程通常能够带来更好的性能和用户体验，但是多线程的程序对于其他程序是不友好的，因为它可能占用了更多的CPU资源。当然，也不是线程越多，程序的性能就越好，因为线程之间的调度和切换也会浪费CPU时间。时下很时髦的[Node.js](https://nodejs.org/)就采用了单线程异步I/O的工作模式。

## **线程的sleep()方法和yield()方法有什么区别？**

答：   
① sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会；yield()方法只会给相同优先级或更高优先级的线程以运行的机会；   
② 线程执行sleep()方法后转入阻塞（blocked）状态，而执行yield()方法后转入就绪（ready）状态；   
③ sleep()方法声明抛出InterruptedException，而yield()方法没有声明任何异常；   
④ sleep()方法比yield()方法（跟操作系统CPU调度相关）具有更好的可移植性。

## **当一个线程进入一个对象的synchronized方法A之后，其它线程是否可进入此对象的synchronized方法B？**

答：不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的synchronized修饰符要求执行方法时要获得对象的锁，如果已经进入A方法说明对象锁已经被取走，那么试图进入B方法的线程就只能在等锁池（**注意不是等待池哦**）中等待对象的锁。

## **请说出与线程同步以及线程调度相关的方法。**

答：   
- wait()：使一个线程处于等待（阻塞）状态，并且释放所持有的对象的锁；   
- sleep()：使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理InterruptedException异常；   
- notify()：唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；   
- notityAll()：唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

**提示：**关于Java多线程和并发编程的问题，建议大家看我的另一篇文章[《关于Java并发编程的总结和思考》](http://blog.csdn.net/jackfrued/article/details/44499227)。

补充：Java 5通过Lock接口提供了显式的锁机制（explicit lock），增强了灵活性以及对线程的协调。Lock接口中定义了加锁（lock()）和解锁（unlock()）的方法，同时还提供了newCondition()方法来产生用于线程之间通信的Condition对象；此外，Java 5还提供了信号量机制（semaphore），信号量可以用来限制对某个共享资源进行访问的线程的数量。在对资源进行访问之前，线程必须得到信号量的许可（调用Semaphore对象的acquire()方法）；在完成对资源的访问后，线程必须向信号量归还许可（调用Semaphore对象的release()方法）。

下面的例子演示了100个线程同时向一个银行账户中存入1元钱，在没有使用同步机制和使用同步机制情况下的执行情况。

银行账户类：

/\*\*

\* 银行账户

\* @author 骆昊

\*

\*/

public class Account {

private double balance; // 账户余额

/\*\*

\* 存款

\* @param money 存入金额

\*/

public void deposit(double money) {

double newBalance = balance + money;

try {

Thread.sleep(10); // 模拟此业务需要一段处理时间

}

catch(InterruptedException ex) {

ex.printStackTrace();

}

balance = newBalance;

}

/\*\*

\* 获得账户余额

\*/

public double getBalance() {

return balance;

}

}

存钱线程类：

/\*\*

\* 存钱线程

\* @author 骆昊

\*

\*/

public class AddMoneyThread implements Runnable {

private Account account; // 存入账户

private double money; // 存入金额

public AddMoneyThread(Account account, double money) {

this.account = account;

this.money = money;

}

@Override

public void run() {

account.deposit(money);

}

}  
测试类：

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Test01 {

public static void main(String[] args) {

Account account = new Account();

ExecutorService service = Executors.newFixedThreadPool(100);

for(int i = 1; i <= 100; i++) {

service.execute(new AddMoneyThread(account, 1));

}

service.shutdown();

while(!service.isTerminated()) {}

System.out.println("账户余额: " + account.getBalance());

}

}

在没有同步的情况下，执行结果通常是显示账户余额在10元以下，出现这种状况的原因是，当一个线程A试图存入1元的时候，另外一个线程B也能够进入存款的方法中，线程B读取到的账户余额仍然是线程A存入1元钱之前的账户余额，因此也是在原来的余额0上面做了加1元的操作，同理线程C也会做类似的事情，所以最后100个线程执行结束时，本来期望账户余额为100元，但实际得到的通常在10元以下（很可能是1元哦）。解决这个问题的办法就是同步，当一个线程对银行账户存钱时，需要将此账户锁定，待其操作完成后才允许其他的线程进行操作，代码有如下几种调整方案：

在银行账户的存款（deposit）方法上同步（synchronized）关键字

/\*\*

\* 银行账户

\* @author 骆昊

\*

\*/

public class Account {

private double balance; // 账户余额

/\*\*

\* 存款

\* @param money 存入金额

\*/

public synchronized void deposit(double money) {

double newBalance = balance + money;

try {

Thread.sleep(10); // 模拟此业务需要一段处理时间

}

catch(InterruptedException ex) {

ex.printStackTrace();

}

balance = newBalance;

}

/\*\*

\* 获得账户余额

\*/

public double getBalance() {

return balance;

}

}

在线程调用存款方法时对银行账户进行同步

/\*\*

\* 存钱线程

\* @author 骆昊

\*

\*/

public class AddMoneyThread implements Runnable {

private Account account; // 存入账户

private double money; // 存入金额

public AddMoneyThread(Account account, double money) {

this.account = account;

this.money = money;

}

@Override

public void run() {

synchronized (account) {

account.deposit(money);

}

}

}

通过Java 5显示的锁机制，为每个银行账户创建一个锁对象，在存款操作进行加锁和解锁的操作

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

/\*\*

\* 银行账户

\*

\* @author 骆昊

\*

\*/

public class Account {

private Lock accountLock = new ReentrantLock();

private double balance; // 账户余额

/\*\*

\* 存款

\*

\* @param money

\* 存入金额

\*/

public void deposit(double money) {

accountLock.lock();

try {

double newBalance = balance + money;

try {

Thread.sleep(10); // 模拟此业务需要一段处理时间

}

catch (InterruptedException ex) {

ex.printStackTrace();

}

balance = newBalance;

}

finally {

accountLock.unlock();

}

}

/\*\*

\* 获得账户余额

\*/

public double getBalance() {

return balance;

}

}

按照上述三种方式对代码进行修改后，重写执行测试代码Test01，将看到最终的账户余额为100元。当然也可以使用Semaphore或CountdownLatch来实现同步。

## **编写多线程程序有几种实现方式？**

答：Java 5以前实现多线程有两种实现方法：一种是继承Thread类；另一种是实现Runnable接口。两种方式都要通过重写run()方法来定义线程的行为，推荐使用后者，因为Java中的继承是单继承，一个类有一个父类，如果继承了Thread类就无法再继承其他类了，显然使用Runnable接口更为灵活。

补充：Java 5以后创建线程还有第三种方式：实现Callable接口，该接口中的call方法可以在线程执行结束时产生一个返回值，代码如下所示：

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

class MyTask implements Callable<Integer> {

private int upperBounds;

public MyTask(int upperBounds) {

this.upperBounds = upperBounds;

}

@Ovrride

public Integer call() throws Exception {

int sum = 0;

for(int i = 1; i <= upperBounds; i++) {

sum += i;

}

return sum;

}

}

class Test {

public static void main(String[] args) throws Exception {

List<Future<Integer>> list = new ArrayList<>();

ExecutorService service = Executors.newFixedThreadPool(10);

for(int i = 0; i < 10; i++) {

list.add(service.submit(new MyTask((int) (Math.random() \* 100))));

}

int sum = 0;

for(Future<Integer> future : list) {

// while(!future.isDone()) ;

sum += future.get();

}

System.out.println(sum);

}

}

## **synchronized关键字的用法？**

答： synchronized关键字可以将对象或者方法标记为同步，以实现对对象和方法的互斥访问，可以用synchronized(对象) { … }定义同步代码块，或者在声明方法时将synchronized作为方法的修饰符。在第60题的例子中已经展示了synchronized关键字的用法。

## **举例说明同步和异步。**

答：如果系统中存在临界资源（资源数量少于竞争资源的线程数量的资源），例如正在写的数据以后可能被另一个线程读到，或者正在读的数据可能已经被另一个线程写过了，那么这些数据就必须进行同步存取（数据库操作中的排他锁就是最好的例子）。当应用程序在对象上调用了一个需要花费很长时间来执行的方法，并且不希望让程序等待方法的返回时，就应该使用异步编程，在很多情况下采用异步途径往往更有效率。事实上，所谓的同步就是指阻塞式操作，而异步就是非阻塞式操作。

## **启动一个线程是调用run()还是start()方法？**

答：启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

## **什么是线程池（thread pool）？**

答：在面向对象编程中，创建和销毁对象是很费时间的，因为创建一个对象要获取内存资源或者其它更多资源。在Java中更是如此，虚拟机将试图跟踪每一个对象，以便能够在对象销毁后进行垃圾回收。所以提高服务程序效率的一个手段就是尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁，这就是”池化资源”技术产生的原因。线程池顾名思义就是事先创建若干个可执行的线程放入一个池（容器）中，需要的时候从池中获取线程不用自行创建，使用完毕不需要销毁线程而是放回池中，从而减少创建和销毁线程对象的开销。   
Java 5+中的Executor接口定义一个执行线程的工具。它的子类型即线程池接口是ExecutorService。要配置一个线程池是比较复杂的，尤其是对于线程池的原理不是很清楚的情况下，因此在工具类Executors面提供了一些静态工厂方法，生成一些常用的线程池，如下所示：   
- newSingleThreadExecutor：创建一个单线程的线程池。这个线程池只有一个线程在工作，也就是相当于单线程串行执行所有任务。如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它。此线程池保证所有任务的执行顺序按照任务的提交顺序执行。   
- newFixedThreadPool：创建固定大小的线程池。每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小。线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。   
- newCachedThreadPool：创建一个可缓存的线程池。如果线程池的大小超过了处理任务所需要的线程，那么就会回收部分空闲（60秒不执行任务）的线程，当任务数增加时，此线程池又可以智能的添加新线程来处理任务。此线程池不会对线程池大小做限制，线程池大小完全依赖于操作系统（或者说JVM）能够创建的最大线程大小。   
- newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。   
- newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

第60题的例子中演示了通过Executors工具类创建线程池并使用线程池执行线程的代码。如果希望在服务器上使用线程池，强烈建议使用newFixedThreadPool方法来创建线程池，这样能获得更好的性能。

## **线程的基本状态以及状态之间的关系？**



**说明：**其中Running表示运行状态，Runnable表示就绪状态（万事俱备，只欠CPU），Blocked表示阻塞状态，阻塞状态又有多种情况，可能是因为调用wait()方法进入等待池，也可能是执行同步方法或同步代码块进入等锁池，或者是调用了sleep()方法或join()方法等待休眠或其他线程结束，或是因为发生了I/O中断。

## **简述synchronized 和java.util.concurrent.locks.Lock的异同？**

答：Lock是Java 5以后引入的新的API，和关键字synchronized相比主要相同点：Lock 能完成synchronized所实现的所有功能；主要不同点：Lock有比synchronized更精确的线程语义和更好的性能，而且不强制性的要求一定要获得锁。synchronized会自动释放锁，而Lock一定要求程序员手工释放，并且最好在finally 块中释放（这是释放外部资源的最好的地方）。

## **Java中如何实现序列化，有什么意义？**

答：序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。   
要实现序列化，需要让一个类实现Serializable接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造一个对象输出流并通过writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过readObject方法从流中读取对象。序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆（可以参考第29题）。

## **Java中有几种类型的流？**

答：字节流和字符流。字节流继承于InputStream、OutputStream，字符流继承于Reader、Writer。在java.io 包中还有许多其他的流，主要是为了提高性能和使用方便。关于Java的I/O需要注意的有两点：一是两种对称性（输入和输出的对称性，字节和字符的对称性）；二是两种设计模式（适配器模式和装潢模式）。另外Java中的流不同于C#的是它只有一个维度一个方向。

**面试题** - 编程实现文件拷贝。（这个题目在笔试的时候经常出现，下面的代码给出了两种实现方案）

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.nio.ByteBuffer;

import java.nio.channels.FileChannel;

public final class MyUtil {

private MyUtil() {

throw new AssertionError();

}

public static void fileCopy(String source, String target) throws IOException {

try (InputStream in = new FileInputStream(source)) {

try (OutputStream out = new FileOutputStream(target)) {

byte[] buffer = new byte[4096];

int bytesToRead;

while((bytesToRead = in.read(buffer)) != -1) {

out.write(buffer, 0, bytesToRead);

}

}

}

}

public static void fileCopyNIO(String source, String target) throws IOException {

try (FileInputStream in = new FileInputStream(source)) {

try (FileOutputStream out = new FileOutputStream(target)) {

FileChannel inChannel = in.getChannel();

FileChannel outChannel = out.getChannel();

ByteBuffer buffer = ByteBuffer.allocate(4096);

while(inChannel.read(buffer) != -1) {

buffer.flip();

outChannel.write(buffer);

buffer.clear();

}

}

}

}

}

**注意：**上面用到Java 7的TWR，使用TWR后可以不用在finally中释放外部资源 ，从而让代码更加优雅。

## **写一个方法，输入一个文件名和一个字符串，统计这个字符串在这个文件中出现的次数。**

import java.io.BufferedReader;

import java.io.FileReader;

public final class MyUtil {

// 工具类中的方法都是静态方式访问的因此将构造器私有不允许创建对象(绝对好习惯)

private MyUtil() {

throw new AssertionError();

}

/\*\*

\* 统计给定文件中给定字符串的出现次数

\*

\* @param filename 文件名

\* @param word 字符串

\* @return 字符串在文件中出现的次数

\*/

public static int countWordInFile(String filename, String word) {

int counter = 0;

try (FileReader fr = new FileReader(filename)) {

try (BufferedReader br = new BufferedReader(fr)) {

String line = null;

while ((line = br.readLine()) != null) {

int index = -1;

while (line.length() >= word.length() && (index = line.indexOf(word)) >= 0) {

counter++;

line = line.substring(index + word.length());

}

}

}

} catch (Exception ex) {

ex.printStackTrace();

}

return counter;

}

}

## **如何用Java代码列出一个目录下所有的文件？**

答：   
如果只要求列出当前文件夹下的文件，代码如下所示：

import java.io.File;

class Test12 {

public static void main(String[] args) {

File f = new File("/Users/Hao/Downloads");

for(File temp : f.listFiles()) {

if(temp.isFile()) {

System.out.println(temp.getName());

}

}

}

}

如果需要对文件夹继续展开，代码如下所示：

import java.io.File;

class Test12 {

public static void main(String[] args) {

showDirectory(new File("/Users/Hao/Downloads"));

}

public static void showDirectory(File f) {

\_walkDirectory(f, 0);

}

private static void \_walkDirectory(File f, int level) {

if(f.isDirectory()) {

for(File temp : f.listFiles()) {

\_walkDirectory(temp, level + 1);

}

}

else {

for(int i = 0; i < level - 1; i++) {

System.out.print("\t");

}

System.out.println(f.getName());

}

}

}

在Java 7中可以使用NIO.2的API来做同样的事情，代码如下所示：

class ShowFileTest {

public static void main(String[] args) throws IOException {

Path initPath = Paths.get("/Users/Hao/Downloads");

Files.walkFileTree(initPath, new SimpleFileVisitor<Path>() {

@Override

public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)

throws IOException {

System.out.println(file.getFileName().toString());

return FileVisitResult.CONTINUE;

}

});

}

}

## **用Java的套接字编程实现一个多线程的回显（echo）服务器。**

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.ServerSocket;

import java.net.Socket;

public class EchoServer {

private static final int ECHO\_SERVER\_PORT = 6789;

public static void main(String[] args) {

try(ServerSocket server = new ServerSocket(ECHO\_SERVER\_PORT)) {

System.out.println("服务器已经启动...");

while(true) {

Socket client = server.accept();

new Thread(new ClientHandler(client)).start();

}

} catch (IOException e) {

e.printStackTrace();

}

}

private static class ClientHandler implements Runnable {

private Socket client;

public ClientHandler(Socket client) {

this.client = client;

}

@Override

public void run() {

try(BufferedReader br = new BufferedReader(new InputStreamReader(client.getInputStream()));

PrintWriter pw = new PrintWriter(client.getOutputStream())) {

String msg = br.readLine();

System.out.println("收到" + client.getInetAddress() + "发送的: " + msg);

pw.println(msg);

pw.flush();

} catch(Exception ex) {

ex.printStackTrace();

} finally {

try {

client.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

}

**注意：**上面的代码使用了Java 7的TWR语法，由于很多外部资源类都间接的实现了AutoCloseable接口（单方法回调接口），因此可以利用TWR语法在try结束的时候通过回调的方式自动调用外部资源类的close()方法，避免书写冗长的finally代码块。此外，上面的代码用一个静态内部类实现线程的功能，使用多线程可以避免一个用户I/O操作所产生的中断影响其他用户对服务器的访问，简单的说就是一个用户的输入操作不会造成其他用户的阻塞。当然，上面的代码使用线程池可以获得更好的性能，因为频繁的创建和销毁线程所造成的开销也是不可忽视的。

下面是一段回显客户端测试代码：

import java.io.BufferedReader;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.net.Socket;

import java.util.Scanner;

public class EchoClient {

public static void main(String[] args) throws Exception {

Socket client = new Socket("localhost", 6789);

Scanner sc = new Scanner(System.in);

System.out.print("请输入内容: ");

String msg = sc.nextLine();

sc.close();

PrintWriter pw = new PrintWriter(client.getOutputStream());

pw.println(msg);

pw.flush();

BufferedReader br = new BufferedReader(new InputStreamReader(client.getInputStream()));

System.out.println(br.readLine());

client.close();

}

}

如果希望用NIO的多路复用套接字实现服务器，代码如下所示。NIO的操作虽然带来了更好的性能，但是有些操作是比较底层的，对于初学者来说还是有些难于理解。

import java.io.IOException;

import java.net.InetSocketAddress;

import java.nio.ByteBuffer;

import java.nio.CharBuffer;

import java.nio.channels.SelectionKey;

import java.nio.channels.Selector;

import java.nio.channels.ServerSocketChannel;

import java.nio.channels.SocketChannel;

import java.util.Iterator;

public class EchoServerNIO {

private static final int ECHO\_SERVER\_PORT = 6789;

private static final int ECHO\_SERVER\_TIMEOUT = 5000;

private static final int BUFFER\_SIZE = 1024;

private static ServerSocketChannel serverChannel = null;

private static Selector selector = null; // 多路复用选择器

private static ByteBuffer buffer = null; // 缓冲区

public static void main(String[] args) {

init();

listen();

}

private static void init() {

try {

serverChannel = ServerSocketChannel.open();

buffer = ByteBuffer.allocate(BUFFER\_SIZE);

serverChannel.socket().bind(new InetSocketAddress(ECHO\_SERVER\_PORT));

serverChannel.configureBlocking(false);

selector = Selector.open();

serverChannel.register(selector, SelectionKey.OP\_ACCEPT);

} catch (Exception e) {

throw new RuntimeException(e);

}

}

private static void listen() {

while (true) {

try {

if (selector.select(ECHO\_SERVER\_TIMEOUT) != 0) {

Iterator<SelectionKey> it = selector.selectedKeys().iterator();

while (it.hasNext()) {

SelectionKey key = it.next();

it.remove();

handleKey(key);

}

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

private static void handleKey(SelectionKey key) throws IOException {

SocketChannel channel = null;

try {

if (key.isAcceptable()) {

ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();

channel = serverChannel.accept();

channel.configureBlocking(false);

channel.register(selector, SelectionKey.OP\_READ);

} else if (key.isReadable()) {

channel = (SocketChannel) key.channel();

buffer.clear();

if (channel.read(buffer) > 0) {

buffer.flip();

CharBuffer charBuffer = CharsetHelper.decode(buffer);

String msg = charBuffer.toString();

System.out.println("收到" + channel.getRemoteAddress() + "的消息：" + msg);

channel.write(CharsetHelper.encode(CharBuffer.wrap(msg)));

} else {

channel.close();

}

}

} catch (Exception e) {

e.printStackTrace();

if (channel != null) {

channel.close();

}

}

}

}

import java.nio.ByteBuffer;

import java.nio.CharBuffer;

import java.nio.charset.CharacterCodingException;

import java.nio.charset.Charset;

import java.nio.charset.CharsetDecoder;

import java.nio.charset.CharsetEncoder;

public final class CharsetHelper {

private static final String UTF\_8 = "UTF-8";

private static CharsetEncoder encoder = Charset.forName(UTF\_8).newEncoder();

private static CharsetDecoder decoder = Charset.forName(UTF\_8).newDecoder();

private CharsetHelper() {

}

public static ByteBuffer encode(CharBuffer in) throws CharacterCodingException{

return encoder.encode(in);

}

public static CharBuffer decode(ByteBuffer in) throws CharacterCodingException{

return decoder.decode(in);

}

}

## **XML文档定义有几种形式？它们之间有何本质区别？解析XML文档有哪几种方式？**

答：XML文档定义分为DTD和Schema两种形式，二者都是对XML语法的约束，其本质区别在于Schema本身也是一个XML文件，可以被XML解析器解析，而且可以为XML承载的数据定义类型，约束能力较之DTD更强大。对XML的解析主要有DOM（文档对象模型，Document Object Model）、SAX（Simple API for XML）和StAX（Java 6中引入的新的解析XML的方式，Streaming API for XML），其中DOM处理大型文件时其性能下降的非常厉害，这个问题是由DOM树结构占用的内存较多造成的，而且DOM解析方式必须在解析文件之前把整个文档装入内存，适合对XML的随机访问（典型的用空间换取时间的策略）；SAX是事件驱动型的XML解析方式，它顺序读取XML文件，不需要一次全部装载整个文件。当遇到像文件开头，文档结束，或者标签开头与标签结束时，它会触发一个事件，用户通过事件回调代码来处理XML文件，适合对XML的顺序访问；顾名思义，StAX把重点放在流上，实际上StAX与其他解析方式的本质区别就在于应用程序能够把XML作为一个事件流来处理。将XML作为一组事件来处理的想法并不新颖（SAX就是这样做的），但不同之处在于StAX允许应用程序代码把这些事件逐个拉出来，而不用提供在解析器方便时从解析器中接收事件的处理程序。

## **你在项目中哪些地方用到了XML？**

答：XML的主要作用有两个方面：数据交换和信息配置。在做数据交换时，XML将数据用标签组装成起来，然后压缩打包加密后通过网络传送给接收者，接收解密与解压缩后再从XML文件中还原相关信息进行处理，XML曾经是异构系统间交换数据的事实标准，但此项功能几乎已经被JSON（JavaScript Object Notation）取而代之。当然，目前很多软件仍然使用XML来存储配置信息，我们在很多项目中通常也会将作为配置信息的硬代码写在XML文件中，Java的很多框架也是这么做的，而且这些框架都选择了[dom4j](http://www.dom4j.org/)作为处理XML的工具，因为Sun公司的官方API实在不怎么好用。

**补充：**现在有很多时髦的软件（如Sublime）已经开始将配置文件书写成JSON格式，我们已经强烈的感受到XML的另一项功能也将逐渐被业界抛弃。

## ****阐述JDBC操作数据库的步骤。****

答：下面的代码以连接本机的Oracle数据库为例，演示JDBC操作数据库的步骤。

### 加载驱动。

Class.forName("oracle.jdbc.driver.OracleDriver");

### 创建连接。

Connection con = DriverManager.getConnection(  
"jdbc:oracle:thin:@localhost:1521:orcl",   
"scott",  
 "tiger");

### 创建语句。

PreparedStatement ps = con.prepareStatement(  
"select \* from emp where sal between ? and ?");

ps.setInt(1, 1000);

ps.setInt(2, 3000);

### 执行语句。

ResultSet rs = ps.executeQuery();

### 处理结果。

while(rs.next()) {

System.out.println(rs.getInt("empno") + " - " + rs.getString("ename"));

}

### 关闭资源。

finally {

if(con != null) {

try {

con.close();

} catch (SQLException e) {

e.printStackTrace();

}

}

}

**提示：**关闭外部资源的顺序应该和打开的顺序相反，也就是说先关闭ResultSet、再关闭Statement、在关闭Connection。上面的代码只关闭了Connection（连接），虽然通常情况下在关闭连接时，连接上创建的语句和打开的游标也会关闭，但不能保证总是如此，因此应该按照刚才说的顺序分别关闭。此外，第一步加载驱动在JDBC 4.0中是可以省略的（自动从类路径中加载驱动），但是我们建议保留。

## **Statement和PreparedStatement有什么区别？哪个性能更好？**

答：与Statement相比，①PreparedStatement接口代表预编译的语句，它主要的优势在于可以减少SQL的编译错误并增加SQL的安全性（减少SQL注射攻击的可能性）；②PreparedStatement中的SQL语句是可以带参数的，避免了用字符串连接拼接SQL语句的麻烦和不安全；③当批量处理SQL或频繁执行相同的查询时，PreparedStatement有明显的性能上的优势，由于数据库可以将编译优化后的SQL语句缓存起来，下次执行相同结构的语句时就会很快（不用再次编译和生成执行计划）。

**补充：**为了提供对存储过程的调用，JDBC API中还提供了CallableStatement接口。存储过程（Stored Procedure）是数据库中一组为了完成特定功能的SQL语句的集合，经编译后存储在数据库中，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。虽然调用存储过程会在网络开销、安全性、性能上获得很多好处，但是存在如果底层数据库发生迁移时就会有很多麻烦，因为每种数据库的存储过程在书写上存在不少的差别。

## **使用JDBC操作数据库时，如何提升读取数据的性能？如何提升更新数据的性能？**

答：要提升读取数据的性能，可以指定通过结果集（ResultSet）对象的setFetchSize()方法指定每次抓取的记录数（典型的空间换时间策略）；要提升更新数据的性能可以使用PreparedStatement语句构建批处理，将若干SQL语句置于一个批处理中执行。

## **在进行数据库编程时，连接池有什么作用？**

答：由于创建连接和释放连接都有很大的开销（尤其是数据库服务器不在本地时，每次建立连接都需要进行TCP的三次握手，释放连接需要进行TCP四次握手，造成的开销是不可忽视的），为了提升系统访问数据库的性能，可以事先创建若干连接置于连接池中，需要时直接从连接池获取，使用结束时归还连接池而不必关闭连接，从而避免频繁创建和释放连接所造成的开销，这是典型的用空间换取时间的策略（浪费了空间存储连接，但节省了创建和释放连接的时间）。池化技术在Java开发中是很常见的，在使用线程时创建线程池的道理与此相同。基于Java的开源数据库连接池主要有：[C3P0](http://sourceforge.net/projects/c3p0/)、[Proxool](http://proxool.sourceforge.net/)、[DBCP](http://commons.apache.org/proper/commons-dbcp/)、[BoneCP](https://github.com/wwadge/bonecp)、[Druid](https://github.com/alibaba/druid)等。

**补充：**在计算机系统中时间和空间是不可调和的矛盾，理解这一点对设计满足性能要求的算法是至关重要的。大型网站性能优化的一个关键就是使用缓存，而缓存跟上面讲的连接池道理非常类似，也是使用空间换时间的策略。可以将热点数据置于缓存中，当用户查询这些数据时可以直接从缓存中得到，这无论如何也快过去数据库中查询。当然，缓存的置换策略等也会对系统性能产生重要影响，对于这个问题的讨论已经超出了这里要阐述的范围。

## **什么是DAO模式？**

答：DAO（Data Access Object）顾名思义是一个为数据库或其他持久化机制提供了抽象接口的对象，在不暴露底层持久化方案实现细节的前提下提供了各种数据访问操作。在实际的开发中，应该将所有对数据源的访问操作进行抽象化后封装在一个公共API中。用程序设计语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口，在逻辑上该类对应一个特定的数据存储。DAO模式实际上包含了两个模式，一是Data Accessor（数据访问器），二是Data Object（数据对象），前者要解决如何访问数据的问题，而后者要解决的是如何用对象封装数据。

## **事务的ACID是指什么？**

答：   
- 原子性(Atomic)：事务中各项操作，要么全做要么全不做，任何一项操作的失败都会导致整个事务的失败；   
- 一致性(Consistent)：事务结束后系统状态是一致的；   
- 隔离性(Isolated)：并发执行的事务彼此无法看到对方的中间状态；   
- 持久性(Durable)：事务完成后所做的改动都会被持久化，即使发生灾难性的失败。通过日志和同步备份可以在故障发生后重建数据。

**补充：**关于事务，在面试中被问到的概率是很高的，可以问的问题也是很多的。首先需要知道的是，只有存在并发数据访问时才需要事务。当多个事务访问同一数据时，可能会存在5类问题，包括3类数据读取问题（脏读、不可重复读和幻读）和2类数据更新问题（第1类丢失更新和第2类丢失更新）。

脏读（Dirty Read）：A事务读取B事务尚未提交的数据并在此基础上操作，而B事务执行回滚，那么A读取到的数据就是脏数据。

| **时间** | **转账事务A** | **取款事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 |  | 查询账户余额为1000元 |
| T4 |  | 取出500元余额修改为500元 |
| T5 | 查询账户余额为500元（脏读） |  |
| T6 |  | 撤销事务余额恢复为1000元 |
| T7 | 汇入100元把余额修改为600元 |  |
| T8 | 提交事务 |  |

不可重复读（Unrepeatable Read）：事务A重新读取前面读取过的数据，发现该数据已经被另一个已提交的事务B修改过了。

| **时间** | **转账事务A** | **取款事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 |  | 查询账户余额为1000元 |
| T4 | 查询账户余额为1000元 |  |
| T5 |  | 取出100元修改余额为900元 |
| T6 |  | 提交事务 |
| T7 | 查询账户余额为900元（不可重复读） |  |

幻读（Phantom Read）：事务A重新执行一个查询，返回一系列符合查询条件的行，发现其中插入了被事务B提交的行。

| **时间** | **统计金额事务A** | **转账事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 | 统计总存款为10000元 |  |
| T4 |  | 新增一个存款账户存入100元 |
| T5 |  | 提交事务 |
| T6 | 再次统计总存款为10100元（幻读） |  |

第1类丢失更新：事务A撤销时，把已经提交的事务B的更新数据覆盖了。

|  |
| --- |
|  |
| **时间** | **取款事务A** | **转账事务B** |
| T1 | 开始事务 |  |
| T2 |  | 开始事务 |
| T3 | 查询账户余额为1000元 |  |
| T4 |  | 查询账户余额为1000元 |
| T5 |  | 汇入100元修改余额为1100元 |
| T6 |  | 提交事务 |
| T7 | 取出100元将余额修改为900元 |  |
| T8 | 撤销事务 |  |
| T9 | 余额恢复为1000元（丢失更新） |  |

第2类丢失更新：事务A覆盖事务B已经提交的数据，造成事务B所做的操作丢失。

| **时间** | **转账事务A** | **取款事务B** |
| --- | --- | --- |
| T1 |  | 开始事务 |
| T2 | 开始事务 |  |
| T3 |  | 查询账户余额为1000元 |
| T4 | 查询账户余额为1000元 |  |
| T5 |  | 取出100元将余额修改为900元 |
| T6 |  | 提交事务 |
| T7 | 汇入100元将余额修改为1100元 |  |
| T8 | 提交事务 |  |
| T9 | 查询账户余额为1100元（丢失更新） |  |

数据并发访问所产生的问题，在有些场景下可能是允许的，但是有些场景下可能就是致命的，数据库通常会通过锁机制来解决数据并发访问问题，按锁定对象不同可以分为表级锁和行级锁；按并发事务锁定关系可以分为共享锁和独占锁，具体的内容大家可以自行查阅资料进行了解。   
直接使用锁是非常麻烦的，为此数据库为用户提供了自动锁机制，只要用户指定会话的事务隔离级别，数据库就会通过分析SQL语句然后为事务访问的资源加上合适的锁，此外，数据库还会维护这些锁通过各种手段提高系统的性能，这些对用户来说都是透明的（就是说你不用理解，事实上我确实也不知道）。ANSI/ISO SQL 92标准定义了4个等级的事务隔离级别，如下表所示：

| **隔离级别** | **脏读** | **不可重复读** | **幻读** | **第一类丢失更新** | **第二类丢失更新** |
| --- | --- | --- | --- | --- | --- |
| READ UNCOMMITED | 允许 | 允许 | 允许 | 不允许 | 允许 |
| READ COMMITTED | 不允许 | 允许 | 允许 | 不允许 | 允许 |
| REPEATABLE READ | 不允许 | 不允许 | 允许 | 不允许 | 不允许 |
| SERIALIZABLE | 不允许 | 不允许 | 不允许 | 不允许 | 不允许 |

需要说明的是，事务隔离级别和数据访问的并发性是对立的，事务隔离级别越高并发性就越差。所以要根据具体的应用来确定合适的事务隔离级别，这个地方没有万能的原则。

## **JDBC中如何进行事务处理？**

答：Connection提供了事务处理的方法，通过调用setAutoCommit(false)可以设置手动提交事务；当事务完成后用commit()显式提交事务；如果在事务处理过程中发生异常则通过rollback()进行事务回滚。除此之外，从JDBC 3.0中还引入了Savepoint（保存点）的概念，允许通过代码设置保存点并让事务回滚到指定的保存点。



## **JDBC能否处理Blob和Clob？**

答： Blob是指二进制大对象（Binary Large Object），而Clob是指大字符对象（Character Large Objec），因此其中Blob是为存储大的二进制数据而设计的，而Clob是为存储大的文本数据而设计的。JDBC的PreparedStatement和ResultSet都提供了相应的方法来支持Blob和Clob操作。下面的代码展示了如何使用JDBC操作LOB：   
下面以MySQL数据库为例，创建一个张有三个字段的用户表，包括编号（id）、姓名（name）和照片（photo），建表语句如下：

create table tb\_user

(

id int primary key auto\_increment,

name varchar(20) unique not null,

photo longblob

);

下面的Java代码向数据库中插入一条记录：

import java.io.FileInputStream;

import java.io.IOException;

import java.io.InputStream;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

class JdbcLobTest {

public static void main(String[] args) {

Connection con = null;

try {

// 1. 加载驱动（Java6以上版本可以省略）

Class.forName("com.mysql.jdbc.Driver");

// 2. 建立连接

con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "123456");

// 3. 创建语句对象

PreparedStatement ps = con.prepareStatement("insert into tb\_user values (default, ?, ?)");

ps.setString(1, "骆昊"); // 将SQL语句中第一个占位符换成字符串

try (InputStream in = new FileInputStream("test.jpg")) { // Java 7的TWR

ps.setBinaryStream(2, in); // 将SQL语句中第二个占位符换成二进制流

// 4. 发出SQL语句获得受影响行数

System.out.println(ps.executeUpdate() == 1 ? "插入成功" : "插入失败");

} catch(IOException e) {

System.out.println("读取照片失败!");

}

} catch (ClassNotFoundException | SQLException e) { // Java 7的多异常捕获

e.printStackTrace();

} finally { // 释放外部资源的代码都应当放在finally中保证其能够得到执行

try {

if(con != null && !con.isClosed()) {

con.close(); // 5. 释放数据库连接

con = null; // 指示垃圾回收器可以回收该对象

}

} catch (SQLException e) {

e.printStackTrace();

}

}

}

}

## **简述正则表达式及其用途。**

答：在编写处理字符串的程序时，经常会有查找符合某些复杂规则的字符串的需要。正则表达式就是用于描述这些规则的工具。换句话说，正则表达式就是记录文本规则的代码。

**说明：**计算机诞生初期处理的信息几乎都是数值，但是时过境迁，今天我们使用计算机处理的信息更多的时候不是数值而是字符串，正则表达式就是在进行字符串匹配和处理的时候最为强大的工具，绝大多数语言都提供了对正则表达式的支持。

## **Java中是如何支持正则表达式操作的？**

答：Java中的String类提供了支持正则表达式操作的方法，包括：matches()、replaceAll()、replaceFirst()、split()。此外，Java中可以用Pattern类表示正则表达式对象，它提供了丰富的API进行各种正则表达式操作，请参考下面面试题的代码。

**面试题：** - 如果要从字符串中截取第一个英文左括号之前的字符串，例如：北京市(朝阳区)(西城区)(海淀区)，截取结果为：北京市，那么正则表达式怎么写？

import java.util.regex.Matcher;

import java.util.regex.Pattern;

class RegExpTest {

public static void main(String[] args) {

String str = "北京市(朝阳区)(西城区)(海淀区)";

Pattern p = Pattern.compile(".\*?(?=\\()");

Matcher m = p.matcher(str);

if(m.find()) {

System.out.println(m.group());

}

}

}

**说明：**上面的正则表达式中使用了懒惰匹配和前瞻，如果不清楚这些内容，推荐读一下网上很有名的[《正则表达式30分钟入门教程》](http://www.jb51.net/tools/zhengze.html)。

## **获得一个类的类对象有哪些方式？**

答：   
- 方法1：类型.class，例如：String.class   
- 方法2：对象.getClass()，例如："hello".getClass()   
- 方法3：Class.forName()，例如：Class.forName("java.lang.String")

## **简述一下你了解的设计模式。**

答：所谓设计模式，就是一套被反复使用的代码设计经验的总结（情境中一个问题经过证实的一个解决方案）。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式使人们可以更加简单方便的复用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。   
在GoF的《Design Patterns: Elements of Reusable Object-Oriented Software》中给出了三类（创建型[对类的实例化过程的抽象化]、结构型[描述如何将类或对象结合在一起形成更大的结构]、行为型[对在不同的对象之间划分责任和算法的抽象化]）共23种设计模式，包括：Abstract Factory（抽象工厂模式），Builder（建造者模式），Factory Method（工厂方法模式），Prototype（原始模型模式），Singleton（单例模式）；Facade（门面模式），Adapter（适配器模式），Bridge（桥梁模式），Composite（合成模式），Decorator（装饰模式），Flyweight（享元模式），Proxy（代理模式）；Command（命令模式），Interpreter（解释器模式），Visitor（访问者模式），Iterator（迭代子模式），Mediator（调停者模式），Memento（备忘录模式），Observer（观察者模式），State（状态模式），Strategy（策略模式），Template Method（模板方法模式）， Chain Of Responsibility（责任链模式）。   
面试被问到关于设计模式的知识时，可以拣最常用的作答，例如：   
- 工厂模式：工厂类可以根据条件生成不同的子类实例，这些子类有一个公共的抽象父类并且实现了相同的方法，但是这些方法针对不同的数据进行了不同的操作（多态方法）。当得到子类的实例后，开发人员可以调用基类中的方法而不必考虑到底返回的是哪一个子类的实例。   
- 代理模式：给一个对象提供一个代理对象，并由代理对象控制原对象的引用。实际开发中，按照使用目的的不同，代理可以分为：远程代理、虚拟代理、保护代理、Cache代理、防火墙代理、同步化代理、智能引用代理。   
- 适配器模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起使用的类能够一起工作。   
- 模板方法模式：提供一个抽象类，将部分逻辑以具体方法或构造器的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法（多态实现），从而实现不同的业务逻辑。   
除此之外，还可以讲讲上面提到的门面模式、桥梁模式、单例模式、装潢模式（Collections工具类和I/O系统中都使用装潢模式）等，反正基本原则就是拣自己最熟悉的、用得最多的作答，以免言多必失。

## **用Java写一个单例类。**

答：   
- 饿汉式单例

public class Singleton {

private Singleton(){}

private static Singleton instance = new Singleton();

public static Singleton getInstance(){

return instance;

}

}

懒汉式单例

public class Singleton {

private static Singleton instance = null;

private Singleton() {}

public static synchronized Singleton getInstance(){

if (instance == null) instance ＝ new Singleton();

return instance;

}

}

**注意：**实现一个单例有两点注意事项，①将构造器私有，不允许外界通过构造器创建对象；②通过公开的静态方法向外界返回类的唯一实例。这里有一个问题可以思考：Spring的IoC容器可以为普通的类创建单例，它是怎么做到的呢？

## **什么是UML？**

答：UML是统一建模语言（Unified Modeling Language）的缩写，它发表于1997年，综合了当时已经存在的面向对象的建模语言、方法和过程，是一个支持模型化和软件系统开发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持。使用UML可以帮助沟通与交流，辅助应用设计和文档的生成，还能够阐释系统的结构和行为。

## **UML中有哪些常用的图？**  答：UML定义了多种图形化的符号来描述软件系统部分或全部的静态结构和动态结构，包括：用例图（use case diagram）、类图（class diagram）、时序图（sequence diagram）、协作图（collaboration diagram）、状态图（statechart diagram）、活动图（activity diagram）、构件图（component diagram）、部署图（deployment diagram）等。在这些图形化符号中，有三种图最为重要，分别是：用例图（用来捕获需求，描述系统的功能，通过该图可以迅速的了解系统的功能模块及其关系）、类图（描述类以及类与类之间的关系，通过该图可以快速了解系统）、时序图（描述执行特定任务时对象之间的交互关系以及执行顺序，通过该图可以了解对象能接收的消息也就是说对象能够向外界提供的服务）。  用例图：  这里写图片描述  类图：  这里写图片描述  时序图：  这里写图片描述

## **用Java写一个冒泡排序。**

答：冒泡排序几乎是个程序员都写得出来，但是面试的时候如何写一个逼格高的冒泡排序却不是每个人都能做到，下面提供一个参考代码：

import java.util.Comparator;

/\*\*

\* 排序器接口(策略模式: 将算法封装到具有共同接口的独立的类中使得它们可以相互替换)

\* @author骆昊

\*

\*/

public interface Sorter {

/\*\*

\* 排序

\* @param list 待排序的数组

\*/

public <T extends Comparable<T>> void sort(T[] list);

/\*\*

\* 排序

\* @param list 待排序的数组

\* @param comp 比较两个对象的比较器

\*/

public <T> void sort(T[] list, Comparator<T> comp);

}

import java.util.Comparator;

/\*\*

\* 冒泡排序

\*

\* @author骆昊

\*

\*/

public class BubbleSorter implements Sorter {

@Override

public <T extends Comparable<T>> void sort(T[] list) {

boolean swapped = true;

for (int i = 1, len = list.length; i < len && swapped; ++i) {

swapped = false;

for (int j = 0; j < len - i; ++j) {

if (list[j].compareTo(list[j + 1]) > 0) {

T temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

swapped = true;

}

}

}

}

@Override

public <T> void sort(T[] list, Comparator<T> comp) {

boolean swapped = true;

for (int i = 1, len = list.length; i < len && swapped; ++i) {

swapped = false;

for (int j = 0; j < len - i; ++j) {

if (comp.compare(list[j], list[j + 1]) > 0) {

T temp = list[j];

list[j] = list[j + 1];

list[j + 1] = temp;

swapped = true;

}

}

}

}

}

## **用Java写一个折半查找。**

答：折半查找，也称二分查找、二分搜索，是一种在有序数组中查找某一特定元素的搜索算法。搜素过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜素过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组已经为空，则表示找不到指定的元素。这种搜索算法每一次比较都使搜索范围缩小一半，其时间复杂度是O(logN)。

import java.util.Comparator;

public class MyUtil {

public static <T extends Comparable<T>> int binarySearch(T[] x, T key) {

return binarySearch(x, 0, x.length- 1, key);

}

// 使用循环实现的二分查找

public static <T> int binarySearch(T[] x, T key, Comparator<T> comp) {

int low = 0;

int high = x.length - 1;

while (low <= high) {

int mid = (low + high) >>> 1;

int cmp = comp.compare(x[mid], key);

if (cmp < 0) {

low= mid + 1;

}

else if (cmp > 0) {

high= mid - 1;

}

else {

return mid;

}

}

return -1;

}

// 使用递归实现的二分查找

private static<T extends Comparable<T>> int binarySearch(T[] x, int low, int high, T key) {

if(low <= high) {

int mid = low + ((high -low) >> 1);

if(key.compareTo(x[mid])== 0) {

return mid;

}

else if(key.compareTo(x[mid])< 0) {

return binarySearch(x,low, mid - 1, key);

}

else {

return binarySearch(x,mid + 1, high, key);

}

}

return -1;

}

}

**说明：**上面的代码中给出了折半查找的两个版本，一个用递归实现，一个用循环实现。需要注意的是计算中间位置时不应该使用(high+ low) / 2的方式，因为加法运算可能导致整数越界，这里应该使用以下三种方式之一：low + (high - low) / 2或low + (high – low) >> 1或(low + high) >>> 1（>>>是逻辑右移，是不带符号位的右移）