**1、面向对象的特征**

- **抽象**：将一类对象的共同特征总结出来构造类。数据抽象和行为抽象。只关注对象有哪些

属性和行为，不关注行为细节。

- **继承：**从已有类（父类（超类、基类））得到信息创建新类（子类（派生类））。封装程序中可变因素的重要手段。

- **封装**：把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。

|  |
| --- |
| 面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。  在类中编写的方法就是对实现细节的封装；  类就是对数据和数据操作的封装。  封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口 |

- **多态性**：允许不同子类型的对象对同一消息作出不同的响应。就是用同样的对象引用调用同样的方法但是做了不同的事情。

分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当A系统访问B系统提供的服务时，B系统有多种提供服务的方式，但一切对A系统来说都是透明的（就像电动剃须刀是A系统，它的供电系统是B系统，B系统可以使用电池供电或者用交流电，甚至还有可能是太阳能，A系统只会通过B类对象调用供电的方法，但并不知道供电系统的底层实现是什么，究竟通过何种方式获得了动力）。

方法重载（overload）实现的是编译时的多态性（也称为前绑定），

方法重写（override）实现的是运行时的多态性（也称为后绑定）。是面向对象最精髓的东西

要实现多态：

1).方法重写（子类继承父类并重写父类中已有的或抽象的方法）；

2).对象造型（用父类型引用引用子类型对象，同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

2、访问修饰符public,private,protected,以及不写（默认）时的区别？

答：



类的成员不写访问修饰时默认default。

默认:对同一包中其他类相当于公开（public），对非同一包中其他类相当于私有（private）。

Protected：对子类相当于公开，对不是同一包中的没有父子关系的类相当于私有。

Java中，外部类的修饰符只能是public或默认，类的成员（包括内部类）的修饰符可是以上四种。

3、String 是最基本的数据类型吗？

答：不是。Java中基本数据类型8个：byte、short、int、long、float、double、char、boolean；其它都是引用类型（reference type），枚举类型也算一种较特殊的引用类型。

4、float f=3.4;是否正确？

答:不正确。3.4是双精度数，将双精度型（double）赋值给浮点型（float）属于下转型（down-casting，也称为窄化）会造成精度损失，因此需要强制类型转换float f =(float)3.4; 或者写成float f =3.4F;。

5、short s1 = 1; s1 = s1 + 1;有错吗?short s1 = 1; s1 += 1;有错吗？

答：对于short s1 = 1; s1 = s1 + 1;由于1是int类型，因此s1+1运算结果也是int 型，需要强制转换类型才能赋值给short型。

而short s1 = 1; s1 += 1;可以正确编译，因为s1+= 1;相当于s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

6、Java有没有goto？

答：goto 是Java中的保留字，在目前版本的Java中没有使用。（根据James Gosling（Java之父）编写的《The Java Programming Language》一书的附录中给出了一个Java关键字列表，其中有goto和const，但是这两个是目前无法使用的关键字，因此有些地方将其称之为保留字，其实保留字这个词应该有更广泛的意义，因为熟悉C语言的程序员都知道，在系统类库中使用过的有特殊意义的单词或单词的组合都被视为保留字）

7、int和Integer有什么区别？

答：Java是一个近乎纯洁的面向对象编程语言，但为了编程的方便还是引入了基本数据类型，但为了能够将这些基本数据类型当成对象操作，Java为每一个基本数据类型都引入了对应的包装类型（wrapper class），int的包装类就是Integer，从Java 5开始引入了自动装箱/拆箱机制，使得二者可以相互转换。

Java 为每个原始类型提供了包装类型：

- 原始类型: byte，short，int，long，float，double， boolean，char

- 包装类型:Byte，Short，Integer，Long，Float，Double ，Boolean，Character

|  |
| --- |
| class AutoUnboxingTest {  public static void main(String[] args) {  Integer a = new Integer(3);  Integer b = 3; // 将3自动装箱成Integer类型  int c = 3;  System.out.println(a == b); // false 两个引用没有引用同一对象  System.out.println(a == c); // true a自动拆箱成int类型再和c比较  }  } |

和自动装箱和拆箱有点关系，代码如下所示：

|  |
| --- |
| public class Test03 {  public static void main(String[] args) {  Integer f1 = 100, f2 = 100, f3 = 150, f4 = 150;  System.out.println(f1 == f2);// true -128-127之间，不会new新的对象  System.out.println(f3 == f4); //false  }  } |

如果不明就里很容易认为两个输出要么都是true要么都是false。首先需要注意的是f1、f2、f3、f4四个变量都是Integer对象引用，所以下面的==运算比较的不是值而是引用。装箱的本质是什么呢？当我们给一个Integer对象赋一个int值的时候，会调用Integer类的静态方法valueOf，如果看看valueOf的源代码就知道发生了什么。

|  |
| --- |
| public static Integer valueOf(int i) {  if (i >= IntegerCache.low && i <= IntegerCache.high)  return IntegerCache.cache[i + (-IntegerCache.low)];  return new Integer(i);  } |

IntegerCache是Integer的内部类，其代码如下所示：

|  |
| --- |
| /\*\*  \* Cache to support the object identity semantics of autoboxing for values between  \* -128 and 127 (inclusive) as required by JLS.  \*  \* The cache is initialized on first usage. The size of the cache  \* may be controlled by the {@code -XX:AutoBoxCacheMax=<size>} option.  \* During VM initialization, java.lang.Integer.IntegerCache.high property  \* may be set and saved in the private system properties in the  \* sun.misc.VM class.  \*/  private static class IntegerCache {  static final int low = -128;  static final int high;  static final Integer cache[];  static {  // high value may be configured by property  int h = 127;  String integerCacheHighPropValue =  sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");  if (integerCacheHighPropValue != null) {  try {  int i = parseInt(integerCacheHighPropValue);  i = Math.max(i, 127);  // Maximum array size is Integer.MAX\_VALUE  h = Math.min(i, Integer.MAX\_VALUE - (-low) -1);  } catch( NumberFormatException nfe) {  // If the property cannot be parsed into an int, ignore it.  }  }  high = h;  cache = new Integer[(high - low) + 1];  int j = low;  for(int k = 0; k < cache.length; k++)  cache[k] = new Integer(j++);  // range [-128, 127] must be interned (JLS7 5.1.7)  assert IntegerCache.high >= 127;  }  private IntegerCache() {}  } |

简单的说，如果整型字面量的值在-128到127之间，那么不会new新的Integer对象，而是直接引用常量池中的Integer对象，所以上面题中f1==f2结果是true，而f3==f4结果是false。

8、&和&&的区别？

答：&运算符有两种用法：(1)按位与；(2)逻辑与。

&&:短路与运算。如&&左边的表达式值是false，右边表达式会被直接短路掉，不会运算。例如在验证用户登录时判定用户名不是null而且不是空字符串，应当写为：username != null &&!username.equals("")，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立，根本不能进行字符串的equals比较，否则会产生NullPointerException异常。

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



答：基本数据类型的变量，对象的引用，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；

程序中字面量如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");

变量str放在栈上，用new创建出来的字符串对象放在堆上，"hello"字面量放在方法区。

方法区有个静态区，专门存放静态变量和静态块。

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);

10、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。

11、switch 是否能作用在byte 上，是否能作用在long 上，是否能作用在String上？

答：在Java 5以前，switch(expr)中，expr只能是byte、short、char、int。

从Java 5开始，Java中引入了枚举类型，expr也可以是enum类型，

Java 7开始，expr还可为字符串（String），但长整型（long）在目前所有的版本中都不可以

12、用最有效率的方法计算2乘以8？

答： 2 << 3（左移3位相当于乘以2的3次方，右移3位相当于除以2的3次方）。

补充：为编写的类重写hashCode方法时，可能会看到如下所示代码，其实不太理解为什么要使用这样的乘法运算来产生哈希码（散列码），而且为什么这个数是个素数，为什么通常选择31这个数？

选择31是因为可用移位和减法运算来代替乘法，从而得到更好的性能。31 \* num 等价于(num << 5) - num，左移5位相当于乘以2的5次方再减去自身就相当于乘以31，现在的VM都能自动完成这个优化。

|  |
| --- |
| public class PhoneNumber {  private int areaCode;  private String prefix;  private String lineNumber;  @Override  public int hashCode() {  final int prime = 31;  int result = 1;  result = prime \* result + areaCode;  result = prime \* result+ ((lineNumber == null) ? 0 : lineNumber.hashCode());  result = prime \* result + ((prefix == null) ? 0 : prefix.hashCode());  return result;  }  @Override  public boolean equals(Object obj) {  if (this == obj)  return true;  if (obj == null)  return false;  if (getClass() != obj.getClass())  return false;  PhoneNumber other = (PhoneNumber) obj;  if (areaCode != other.areaCode)  return false;  if (lineNumber == null) {  if (other.lineNumber != null)  return false;  } else if (!lineNumber.equals(other.lineNumber))  return false;  if (prefix == null) {  if (other.prefix != null)  return false;  } else if (!prefix.equals(other.prefix))  return false;  return true;  }  } |

13、数组有没有length()方法？String有没有length()方法？

答：数组没有length()方法，有length 的属性。

String 有length()方法。JavaScript中，获得字符串的长度是通过length属性得到的.

14、在Java中，如何跳出当前的多重嵌套循环？

答：在最外层循环前加一个标记如A，然后用break A;可以跳出多重循环。

（Java中支持带标签的break和continue语句，作用有点类似于C和C++中的goto语句，但是就像要避免使用goto一样，应该避免使用带标签的break和continue，因为它不会让你的程序变得更优雅，很多时候甚至有相反的作用，所以这种语法其实不知道更好）

15、构造器（constructor）是否可被重写（override）？

答：构造器不能被继承，因此不能被重写，但可被重载。

构造函数不是成员函数。是用来造唯一的东西的。不能用一个构造函数既造爸爸，又造儿子。这样身份就乱了。

16、两个对象值相同(x.equals(y) == true)，但却可有不同的hashcode，这句话对不对？

答：不对，如果两个对象x和y满足x.equals(y) == true，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，（《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）：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）

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：

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Object对象替换为其他类型，重写时不要忘掉@Override注解。

17、是否可以继承String类？

答：String 类是final类，不可被继承。

补充：继承String本身就是一个错误的行为，对String类型最好的重用方式是关联关系（Has-A）和依赖关系（Use-A）而不是继承关系（Is-A）。

18、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#中可通过传引用或传输出参数来改变传入的参数的值。

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（包装类）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。

19、String和StringBuilder、StringBuffer的区别？

答：可储存和操作字符串。

String是只读字符串，引用的字符串内容不能被改变。

StringBuffer/StringBuilder类表示的字符串对象可直接进行修改。

StringBuilder和StringBuffer方法完全相同，区别在于它是在单线程环境下使用的，因为它的所有方面都没有被synchronized修饰，因此它的效率也比StringBuffer要高。

题1 - 什么情况下用+运算符进行字符串连接比调用StringBuffer/StringBuilder对象的append方法连接字符串性能更好？

答案:如使用少量的字符串操作，使用 (+运算符)连接字符串；

如果频繁的对大量字符串进行操作，则使用

(1)全局变量或者需要多线程支持则使用StringBuffer；

(2)局部变量或者单线程不涉及线程安全则使有StringBuilder。

题2 - 请说出下面程序的输出。

|  |
| --- |
| class StringEqualTest {  public static void main(String[] args) {  String s1 = "Programming";  String s2 = new String("Programming");  String s3 = "Program";  String s4 = "ming";  String s5 = "Program" + "ming";  String s6 = s3 + s4;  System.out.println(s1 == s2);  System.out.println(s1 == s5);  System.out.println(s1 == s6);  System.out.println(s1 == s6.intern());  System.out.println(s2 == s2.intern());  }  } |

补充：

1. String对象的intern方法会得到字符串对象在常量池中对应的版本的引用（如果常量池中有一个字符串与String对象的equals结果是true），如果常量池中没有对应的字符串，则该字符串将被添加到常量池中，然后返回常量池中字符串的引用；

2. 字符串的+操作其本质是创建了StringBuilder对象进行append操作，然后将拼接后的StringBuilder对象用toString方法处理成String对象，这一点可以用javap -c StringEqualTest.class命令获得class文件对应的JVM字节码指令就可以看出来。

20、重载（Overload）和重写（Override）的区别。重载的方法能否根据返回类型进行区分？

答：(1)都是实现多态的方式

(2)重载:编译时的多态性

重写:运行时的多态性

(3)重载发生在一个类中，同名方法有不同的参数列表（参数类型、参数个数或二者都不同）

重写发生在子类与父类之间，在子类中定义某方法与其父类有相同的名称和参数及返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。

(如需父类中原有的方法，可使用super关键字，该关键字引用了当前类的父类)

重载对返回类型没有特殊的要求。

* "为什么不能根据返回类型来区分重载"?

对于重载方法，不要试图通过给予不同的返回值类型（参数列表完全相同）加以区分。因为方法在调用时可以不将返回值赋值给一个对应的变量，这样就没有特征区分。也不要试图通过在调用这样的重载方法时赋值给对应的变量加以区分，因为问题是出现在重载方法定义上，而不是调用上。总而言之，Java不允许仅仅利用返回值的不同来区分重载方法。

21、描述一下JVM加载class文件的原理机制？

答：JVM中类的装载由类加载器（ClassLoader）和它的子类来实现.



类加载器是一个重要的Java运行时系统组件，负责在运行时查找和装入类文件中的类。

因Java的跨平台性，编译后的Java源程序并不是一个可执行程序，而是一个或多个类文件。当Java程序需要使用某个类时，JVM会确保这个类已经被加载、连接（验证、准备和解析）和初始化。

类的加载是指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，然后产生与所加载类对应的Class对象。加载完成后，Class对象还不完整，所以此时的类还不可用。

类被加载后就进入连接阶段，包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。

最后JVM对类进行初始化：

1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；

2)如果类中存在初始化语句，就依次执行这些初始化语句。

类的加载由类加载器完成的，类加载器包括：

* 根加载器（BootStrap）
* 扩展加载器（Extension）
* 系统加载器（System）
* 用户自定义类加载器（java.lang.ClassLoader的子类）。

从Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。更好的保证了Java平台的安全性。在该机制中，JVM自带的Bootstrap是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM不会向Java程序提供对Bootstrap的引用。

说明：

• Bootstrap：一般用本地代码实现，负责加载JVM基础核心类库（rt.jar）；

• Extension：从java.ext.dirs系统属性所指定的目录中加载类库，它的父加载器是Bootstrap；

• System：又叫应用类加载器，其父类是Extension。是应用最广泛的类加载器。从环境变量classpath或系统属性java.class.path所指定的目录中记载类，是用户自定义加载器的默认父加载器。



22、char 型变量中能不能存贮一个中文汉字，为什么？

答：能. Java使用Unicode编码（不选择任何特定的编码，直接使用字符在字符集中的编号，这是统一的唯一方法），所有文字都用2字节来表示。一个char类型占2字节（16比特），中文汉字(2字节)，所以放一个中文是没问题的。

补充：使用Unicode意味着字符在JVM内部和外部有不同的表现形式，在JVM内部都是Unicode，当这个字符被从JVM内部转移到外部时（如存入文件系统中），需进行编码转换。

所以Java中有字节流和字符流，以及在字符流和字节流之间进行转换的转换流，如InputStreamReader和OutputStreamReader，这两个类是字节流和字符流之间的适配器类，承担了编码转换的任务；。

23、抽象类（abstract class）和接口（interface）有什么异同？

答：

**相同点:**

* 抽象类和接口都不能够实例化，但可定义抽象类和接口类型的引用。
* 一个类如果继承了某抽象类或实现了某接口, 都需对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。

**不同点：**

* 接口比抽象类更加抽象，因为抽象类中可以定义构造器，可以有抽象方法和具体方法，而接口中不能定义构造器而且其中的方法全都是抽象方法。
* 抽象类中成员可是private、默认、protected、public，而接口中的成员全都是public的。
* 抽象类中可定义成员变量，而接口中定义的成员变量实际上都是常量。
* 有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法。

24、静态嵌套类(Static Nested Class)和内部类（Inner Class）的不同？

答：Static Nested Class是被声明为静态(static)的内部类，可不依赖于外部类实例被实例化。

而通常的内部类需要在外部类实例化后才能实例化，其语法看起来挺诡异的，如下所示。

|  |
| --- |
| /\*\*  \* 扑克类（一副扑克）  \*/  public class Poker {  private static String[] suites = {"黑桃", "红桃", "草花", "方块"};  private static int[] faces = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};  private Card[] cards;  /\*\*  \* 构造器  \*  \*/  public Poker() {  cards = new Card[52];  for(int i = 0; i < suites.length; i++) {  for(int j = 0; j < faces.length; j++) {  cards[i \* 13 + j] = new Card(suites[i], faces[j]);  }  }  }  /\*\*  \* 洗牌 （随机乱序）  \*/  public void shuffle() {  for(int i = 0, len = cards.length; i < len; i++) {  int index = (int) (Math.random() \* len);  Card temp = cards[index];  cards[index] = cards[i];  cards[i] = temp;  }  }  /\*\*  \* 发牌  \* @param index 发牌的位置  \*/  public Card deal(int index) {  return cards[index];  }  /\*\*  \* 卡片类（一张扑克）  \* [内部类]  \*  \*/  public class Card {  private String suite; // 花色  private int face; // 点数  public Card(String suite, int face) {  this.suite = suite;  this.face = face;  }  @Override  public String toString() {  String faceStr = "";  switch(face) {  case 1: faceStr = "A"; break;  case 11: faceStr = "J"; break;  case 12: faceStr = "Q"; break;  case 13: faceStr = "K"; break;  default: faceStr = String.valueOf(face);  }  return suite + faceStr;  }  }  }  测试代码：  class PokerTest {  public static void main(String[] args) {  Poker poker = new Poker();  poker.shuffle(); // 洗牌  Poker.Card c1 = poker.deal(0); // 发第一张牌  // 对于非静态内部类Card  // 只有通过其外部类Poker对象才能创建Card对象  Poker.Card c2 = poker.new Card("红心", 1); // 自己创建一张牌  System.out.println(c1); // 洗牌后的第一张  System.out.println(c2); // 打印: 红心A  }  } |

下面的代码哪些地方会产生编译错误？

|  |
| --- |
| public class Outer {  class Inner{}  //外部类的静态方法中不能new内部类的实例对象  public static void foo() {  //new Inner();//报错:没有可以访问外部类型的外部实例。  //new Outer().new Inner();//正确  }    public void bar() {  new Inner();  }    public static void main(String[] args) {  //new Inner();//报错:没有可以访问外部类型的外部实例。  new Outer().new Inner();//正确  }  } |

注意：Java中非静态内部类对象的创建要依赖其外部类对象，上面题中foo和main方法都是静态方法，静态方法中没有this，也就是说没有所谓的外部类对象，因此无法创建内部类对象，如果要在静态方法中创建内部类对象，可以这样做：

new Outer().new Inner();

25、Java 中会存在内存泄漏吗，请简单描述。

答：

理论上Java因有垃圾回收机制（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）；

实际开发中，可能会存在无用但可达的对象，不能被GC回收，也会导致内存泄露。

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，甚至可以通过单元测试。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。

26、抽象的（abstract）方法是否可同时是静态的（static）,是否可同时是本地方法（native），是否可同时被synchronized修饰？

答：都不能。

抽象方法需要子类重写，而静态的方法是无法被重写的，因此二者是矛盾的。

（因静态的可用类名调用，而调用一个抽象方法没有意义）

本地方法是由本地代码（如C代码）实现的方法，而抽象方法是没有实现的，也是矛盾的。synchronized和方法的实现细节有关，抽象方法不涉及实现细节，因此也是相互矛盾的。

|  |
| --- |
| 重写只能适用于可观察的实例方法.不能用于静态方法和final、private.  static、final、private方法本身都是编译期绑定的（也叫前期绑定），不存在多态，是在还没有运行的时候，程序在编译器里面就知道该调用哪个类的哪个方法了，而其他可观察的普通方法的绑定是在运行的时候根据具体的对象决定的，是可被继承的，有了多态而造成了不确定性 |

27、阐述静态变量和实例变量的区别。

答：

* 静态变量：static修饰的变量，也称类变量，属于类，不属于类的任何一个对象，一个类不管创建多少个对象，静态变量在内存中有且仅有一个拷贝；

静态变量可以实现让多个对象共享内存。

* 实例变量依存于某一实例，需要先创建对象然后通过对象才能访问到它。

补充：在Java开发中，上下文类和工具类中通常会有大量的静态成员。

28、是否可以从一个静态（static）方法内部发出对非静态（non-static）方法的调用？

答：不可以，静态方法只能访问静态成员，因为非静态方法的调用要先创建对象，在调用静态方法时可能对象并没有被初始化。

29、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，可实现真正的深度克隆

|  |
| --- |
| 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  \*/  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。

30、GC是什么？为什么要有GC？

答：GC（垃圾收集），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

GC功能可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，可调：System.gc() 或Runtime.getRuntime().gc() ，但JVM可屏蔽掉显示的垃圾回收调用。

垃圾回收可有效的防止内存泄露，有效的使用可使用的内存。

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。

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

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）：对象最初诞生的区域，对大多数对象来说，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

|  |
| --- |
| jvm参数在哪里设置，分情况：  1、集成开发环境下启动并使用JVM，如eclipse需要修改根目录文件eclipse.ini；  2、Windows服务器下安装版Tomcat，可使用Tomcat7w.exe工具（tomcat目录下）和直接修改注册表两种方式修改Jvm参数；  3、Windows服务器解压版Tomcat注册Windows服务，方法同上；  4、解压版本的Tomcat, 通过startup.bat启动tomcat加载配置的，在tomcat 的bin 下catalina.bat 文件内添加；  5、Linux服务器Tomcat设置JVM，修改TOMCAT\_HOME/bin/catalina.sh； |

31、String s = new String("xyz");创建了几个字符串对象？

答：两个对象，一个是静态区(方法区内)的"xyz"，一个是用new创建在堆上的对象。

32、接口是否可继承（extends）接口？抽象类是否可实现（implements）接口？抽象类是否可继承具体类（concrete class）？

答：

* 接口可继承接口，而且支持多重继承。
* 抽象类可实现(implements)接口

|  |
| --- |
| 当只想实现接口中的个别方法（不是所有方法）时，可先写一个抽象类来实现该接口，并实现除了你想要的方法之外的所有方法（方法体为空）。  接着再用你的类继承这个抽象类，这个类中就只用实现你需要的方法了，就可达到需要。但是，如果你直接实现接口的话，你就需要实现接口的所有方法。 |

* 抽象类可继承具体类也可继承抽象类。

|  |  |  |  |
| --- | --- | --- | --- |
| 问： 抽象类是否可继承实体类 (concrete class)  答： 可以。前提是实体类必须有明确的构造函数  Object就是个实体类，java的API里，抽象类条目里都明确写着直接或间接继承自Object。   |  | | --- | | class A{}abstract class B extends A{} |   结果完全正常，编译通过。  涉及到两个基础知识：  1.所有的class必须有构造方法，如没有在代码里声明，系统会自动生成公有无参构造方法。而只要自己声明了，无论有参无参，私有公有，系统就不再帮你生成默认的无参构造器。  2.所有的子类构造器都要求在第一行代码中调用父类构造器，如果不写，系统默认去调用父类的无参构造器。  所以，如果把系统默认配给的方法也算进去，class A{}的代码实际上是   |  | | --- | | class A{  public A(){}  }  B继承 A 的时候，则是  abstract class B extends A{  public B(){  super();  }  } |   假如：   |  | | --- | | class A{  private A(){}  } |   系统不再给你默认无参构造器， B的构造器根据（2）中的规则去调用super()，却找不到A的无参构造器，导致abstract class B extends A{} 编译不能通过。   * “实体类必须有明确的构造函数”的含义：   1.没写构造器的，那是拥有默认无参公有构造函数的  2.写了子类可访问的无参构造器的，也是一样，子类里可以什么都不写，用默认机制调用。  3.写了有参构造器却没写无参构造器的，父类里没有子类可访问的无参构造器，子类必须在子类构造器里的第一句写明，调用父类有参构造器，并把参数传进去。  4.声明为final的以及所有构造器都不在子类访问权限之内的类无法继承  只要是在类的继承中，无论抽象还是实体，都需符合这个规则。。  “可继承，但和实体类的继承一样，也要求父类可继承，并且拥有子类可访问到的构造器。 |

33、一个".java"源文件中是否可以包含多个类（不是内部类）？有什么限制？

答：可以。

但一个源文件中最多只能有一个公开类（public class）

且文件名必须同公开类名。

34、Anonymous Inner Class(匿名内部类)是否可以继承其它类？是否可以实现接口？

答：可继承其他类或实现其他接口，在Swing编程和Android开发中常用此方式来实现事件监听和回调。

35、内部类可引用它的包含类（外部类）的成员吗？有没有什么限制？

答：一个内部类对象可访问创建它的外部类对象的成员，包括私有成员。

36、Java 中的final关键字有哪些用法？

答：

(1)修饰类：表示该类不能被继承；

(2)修饰方法：表示方法不能被重写；

(3)修饰变量：表示变量只能一次赋值以后值不能被修改（常量）。

37、指出下面程序的运行结果。

|  |
| --- |
| class A {  static {  System.out.print("1");  }  public A() {  System.out.print("2");  }  }  class B extends A{  static {  System.out.print("a");  }  public B() {  System.out.print("b");  }  }  public class Hello {  public static void main(String[] args) {  A ab = new B();  ab = new B();  }  } |

答：执行结果：1a2b2b。

创建对象时构造器的调用顺序是：

先初始化静态成员，然后调用父类构造器，再初始化非静态成员，最后调用自身构造器。

38.1如何将字符串转换为基本数据类型？

* 基本数据类型 a=基本数据类型包装类.parse基本数据类型(String str);

|  |
| --- |
| str="123";  int parseInt = Integer.parseInt(str); |

38.2 如何将基本数据类型转换为字符串？

答： 基本数据类型 + ""

String.valueOf(Object)

39、如何实现字符串的反转及替换？

答：可自己写实现，也可使用String或StringBuffer/StringBuilder中方法。

用递归实现字符串反转：

|  |
| --- |
| public static String reverse(String originStr) {  if(originStr == null || originStr.length() <= 1) {  return originStr;  }  return reverse(originStr.substring(1)) + originStr.charAt(0);  } |

40、怎样将GB2312编码的字符串转换为ISO-8859-1编码的字符串？

答：代码如下所示：

|  |
| --- |
| String s1 = "你好";  String s2 = new String(s1.getBytes("GB2312"), "ISO-8859-1"); |

41、日期和时间：

答：

* 问题1：如何取得年月日、小时分钟秒？

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：如何取得从1970年1月1日0时0分0秒到现在的毫秒数？

//当前时间，以从历元至现在所经过的毫秒数形式。

//历元（即格林威治标准时间 1970 年 1 月 1 日的 00:00:00.000，格里高利历）

Calendar.getInstance().getTimeInMillis();

System.currentTimeMillis(); //自1970年1月1日0时起的毫秒数

|  |
| --- |
| 补充:  public Date() {  this(System.currentTimeMillis());  } |

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

问题3：- 如何取得某月的最后一天？

Calendar time = Calendar.getInstance();

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

问题4：如何格式化日期？

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。

42、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);  }  } |

43、比较一下Java和JavaSciprt。

答：

* 两个公司开发的不同产品。Java 是原Sun Microsystems公司推出的面向对象的程序设计语言，特别适合于互联网应用程序开发；而JavaScript是Netscape公司的产品，为了扩展Netscape浏览器的功能而开发的一种可以嵌入Web页面中运行的基于对象和事件驱动的解释性语言。JavaScript的前身是LiveScript；而Java的前身是Oak语言。
* 基于对象和面向对象：Java是一种真正的面向对象的语言，即使是开发简单的程序，必须设计对象；JavaScript是种脚本语言，它可以用来制作与网络无关的，与用户交互作用的复杂软件。它是一种基于对象（Object-Based）和事件驱动（Event-Driven）的编程语言，因而它本身提供了非常丰富的内部对象供设计人员使用。
* 解释和编译：

Java的源代码在执行前，必须经过编译。

JavaScript:解释性，源代码不需编译，由浏览器解释执行。

* 强类型变量和类型弱变量：

Java:强类型变量检查，即所有变量在编译之前必须作声明；

JavaScript:弱类型，在使用变量前可不作声明，JS的解释器在运行时检查推断其数据类型。

* 代码格式不一样。

补充：其实Java和JavaScript最重要的区别是一个是静态语言，一个是动态语言。

目前的编程语言的发展趋势是函数式语言和动态语言。

在Java中类（class）是一等公民，而JavaScript中函数（function）是一等公民，因此JavaScript支持函数式编程，可以使用Lambda函数和闭包（closure），

当然Java 8也开始支持函数式编程，提供了对Lambda表达式以及函数式接口的支持。

44、什么时候用断言（assert）？

答：用于保证程序最基本、关键的正确性。

断言检查通常在开发和测试时开启。

为了保证程序的执行效率，在软件发布后断言检查通常是关闭的。

断言是一个包含布尔表达式的语句，在执行这个语句时假定该表达式为true；

如果表达式的值为false，那么系统会报告一个AssertionError。

断言使用如下面的代码所示：

|  |
| --- |
| assert(a > 0); // throws an AssertionError if a <= 0 |

断言可有两种形式：

|  |
| --- |
| assert Expression1;  assert Expression1 : Expression2 ; |

Expression1 应该总是产生一个布尔值。

Expression2可是得出一个值的任意表达式;这个值用于生成显示更多调试信息的字符串消息。

要在运行时启用断言，可以在启动JVM时使用-enableassertions或者-ea标记。

要在运行时选择禁用断言，可以在启动JVM时使用-da或者-disableassertions标记。

要在系统类中启用或禁用断言，可使用-esa或-dsa标记。

还可以在包的基础上启用或者禁用断言。

注意：断言不应该以任何方式改变程序的状态。简单的说，如果希望在不满足某些条件时阻止代码的执行，就可以考虑用断言来阻止它。

|  |  |
| --- | --- |
| （一）首先明确：  Assert是jdk1.4引入的。  默认关闭。  只适用于复杂的调试过程。  一般用于程序执行结果的判断，不要让断言处理业务流程。  （二）判断eclipse是否开启了断言，代码如下：   |  | | --- | | public class AssertTest {  public static void main(String[] args) {  boolean isOpen = false;  assert isOpen=true; //如果开启了断言，会将isOpen的值改为true  System.out.println(isOpen);//打印是否开启了断言  }  } |   如果打印 true说明已经启用了断言，如果为 false 则没有启用断言。  （三）eclipse中开启断言  选择菜单：Run ---> Run Configurations...---> 选择 Arguments 选项卡  在 VM arguments 文本框中输入： -ea 注意：中间没有空格，如果输入 -da 表示禁止断言。  然后关闭该窗口，然后保存就开启了断言。  (四) 断言使用  assert关键字语法很简单，有两种用法：  1、assert <boolean表达式>  如果<boolean表达式>为true，则程序继续执行。  如果为false，则程序抛出AssertionError，并终止执行。  2、assert <boolean表达式> : <错误信息表达式>  如果<boolean表达式>为true，则程序继续执行。  如果为false，则程序抛出java.lang.AssertionError，并输入<错误信息表达式>  四、陷阱  assert用法简单，但使用assert往往会让你陷入越来越深的陷阱中。应避免使用。  原因：  1、assert关键字需要在运行时候显式开启才能生效，否则断言就没有意义。  主流的Java IDE工具默认都没有开启-ea断言检查功能。  并且，对于Java Web应用，程序代码都是部署在容器里面，没法直接去控制程序的运行，如果一定要开启-ea的开关，则需要更改Web容器的运行配置参数。这对程序的移植和部署都带来很大的不便。  2、用assert代替if是陷阱之二。assert的判断和if语句差不多，但两者的作用有着本质区别：  assert本意上是为测试调试程序时使用的，但如果不小心用assert来控制了程序的业务流程，  那在测试调试结束后去掉assert关键字就意味着修改了程序的正常的逻辑。  3、assert断言失败将面临程序的退出。这在一个生产环境下的应用是绝不能容忍的。  一般都是通过异常处理来解决程序中潜在错误。但使用断言就很危险，一旦失败系统就挂了。  五、对assert的思考  assert既然是为了调试测试程序用，不在正式生产环境下用，那应该考虑更好的测试JUint来代替其做用，  JUint相对assert关键的所提供的功能是有过之而无不及。  当然完全可以通过IDE debug来进行调试测试。  应避免在Java中用assert关键字，除非哪一天Java默认支持开启-ea的开关，这时可考虑。 |

45、Error和Exception有什么区别？

答：Error表系统级的错误和程序不必处理的异常，是不是不可能恢复但很困难的情况下的一种严重问题；比如内存溢出，不可能指望程序能处理这样的情况；

Exception表需要捕捉或需要程序进行处理的异常，是一种设计或实现问题；就是说，它表示如果程序运行正常，从不会发生的情况。

面试题：2005年摩托罗拉的面试问题

“If a process reports a stack overflow run-time error, what’s the most possible cause?”，

a. lack of memory;

b. write on an invalid memory space;

c. recursive function calling;

d. array index out of boundary.

Java程序在运行时也可能会遭遇StackOverflowError，是一个无法恢复的错误，只能重新修改代码了

答案是c。

如果写了不能迅速收敛的递归，则很有可能引发栈溢出的错误，如下所示：

|  |
| --- |
| class StackOverflowErrorTest {  public static void main(String[] args) {  main(null);  }  } |

结果: Exception in thread "main" java.lang.StackOverflowError

提示：用递归编写程序时一定要牢记两点：

1. 递归公式；

2. 收敛条件（什么时候就不再继续递归）。

46、try{}里有一个return语句，那么紧跟在这个try后的finally{}里的代码会不会被执行，什么时候被执行，在return前还是后?

答：会执行，在方法返回调用者前执行。

注意：在finally中改变返回值的做法是不好的，因为如果存在finally代码块，try中的return语句不会立马返回调用者，而是记录下返回值待finally代码块执行完毕之后, 再向调用者返回其值，然后如果在finally中修改了返回值，就会返回修改后的值。

显然，在finally中返回或者修改返回值会对程序造成很大的困扰

C#中直接用编译错误的方式来阻止程序员干这种龌龊的事情

Java中也可通过提升编译器的语法检查级别来产生警告或错误，强烈建议将此项设置为编译错误。

47、Java语言如何进行异常处理，关键字：throws、throw、try、catch、finally分别如何使用？

答：Java通过面向对象的方法进行异常处理，把各种不同的异常进行分类，并提供良好接口。

在Java中，每个异常都是一个对象，是Throwable类或其子类的实例。当一个方法出现异常后便抛出一个异常对象，该对象中包含有异常信息，调用这个对象的方法可以捕获到这个异常并可以对其进行处理。

Java的异常处理通过5个关键词来实现：try、catch、throw、throws和finally。

用try来执行一段程序，如系统会抛出（throw）一个异常对象，可通过它的类型来捕获（catch）它，或通过总是执行代码块（finally）来处理；

try用来指定一块预防所有异常的程序；

catch子句紧跟在try块后面，用来指定想要捕获的异常的类型；

throw语句用来明确地抛出一个异常；

throws用来声明一个方法可能抛出的各种异常（当然声明异常时允许无病呻吟）；

finally为确保一段代码不管发生什么异常状况都要被执行；

try语句可以嵌套，每当遇到一个try语句，异常的结构就会被放入异常栈中，直到所有的try语句都完成。如果下一级的try语句没有对某种异常进行处理，异常栈就会执行出栈操作，直到遇到有处理这种异常的try语句或者最终将异常抛给JVM。

48、运行时异常与受检异常有何异同？

答：异常表示程序运行过程中可能出现的非正常状态

运行时异常表虚拟机的通常操作中可能遇到的异常，是常见运行错误，只要程序设计没有问题通常就不会发生。

受检异常跟程序运行的上下文环境有关，即使程序设计无误，仍然可能因使用的问题而引发。Java编译器要求方法必须声明抛出可能发生的受检异常，但并不要求必须声明抛出未被捕获的运行时异常。

异常和继承一样，是面向对象程序设计中经常被滥用的东西

异常的使用指导原则：

- 不要将异常处理用于正常的控制流（设计良好的API不应该强迫它的调用者为了正常的控制流而使用异常）

- 对可恢复的情况使用受检异常，对编程错误使用运行时异常

- 避免不必要的使用受检异常（可以通过一些状态检测手段来避免异常的发生）

- 优先使用标准的异常

- 每个方法抛出的异常都要有文档

- 保持异常的原子性

- 不要在catch中忽略掉捕获到的异常

49、列出一些你常见的运行时异常？

答：

- ArithmeticException（算术异常）

- ClassCastException （类转换异常）

- NullPointerException （空指针异常）

- IllegalArgumentException （非法参数异常）

- IndexOutOfBoundsException （下标越界异常）

- SecurityException （安全异常）

50、阐述final、finally、finalize的区别。

答：

* final

三种用法：类被声明为final，不能被继承。

变量声明为final，在使用中不被改变，必须在声明时给定初值，引用中只读不可修改。

声明为final的方法不能在子类中被重写。

* finally：

放在try…catch…后面构造总是执行代码块，意味着程序无论正常执行还是发生异常，这里的代码只要JVM不关闭都能执行，可将释放外部资源的代码写在finally块中。

* finalize

Object类中定义的方法. 在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。由垃圾收集器在销毁对象时调用，通过重写finalize()方法可整理系统资源者执行其他清理工作。

51、类ExampleA继承Exception，类ExampleB继承ExampleA。

有如下代码片断：

|  |
| --- |
| try {  throw new ExampleB("b")  } catch（ExampleA e）{  System.out.println("ExampleA");  } catch（Exception e）{  System.out.println("Exception");  } |

请问执行此段代码的输出是什么？

答：输出：ExampleA。（根据里氏代换原则[能使用父类型的地方一定能使用子类型]，

抓取ExampleA类型异常的catch块能够抓住try块中抛出的ExampleB类型的异常）

52、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，List是线性结构的容器，适用于按数值索引访问元素的情形。

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

答：ArrayList 和Vector都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，都允许直接按序号索引元素，但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中方法由于添加了synchronized修饰，因此是线程安全，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属遗留容器（Java早期版本中提供的，Hashtable、Dictionary、BitSet、Stack、Properties都是遗留容器），已不推荐使用，但由于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Properties和Stack类设计上有严重问题

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，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。

54、Collection和Collections的区别？

答：Collection是一个接口，是Set、List等容器的父接口；

Collections是一个工具类，提供了一系列的静态方法来辅助容器操作，包括对容器的搜索、排序、线程安全化等等。

55、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

基于哈希存储的理论存取时间复杂度为O(1)

而基于排序树版本在插入或删除元素时会按元素或元素的键（key）构成排序树从而达到排序和去重的效果。

56、TreeMap和TreeSet在排序时如何比较元素？Collections.sort()方法如何比较元素？

答：

TreeSet要求存放对象所属的类必须实现Comparable接口,通过compareTo()比较元素大小。

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实现Comparable接口以实现元素比较；
* 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]  }  } |

57、Thread类的sleep()方法和对象的wait()方法都可让线程暂停执行，有什么区别?

答：

sleep()方法是线程类的静态方法，调用此方法会让当前线程暂停执行指定时间，将执行机会（CPU）让给其他线程，但对象的锁依然保持，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，如果线程重新获得对象的锁就可进入就绪状态。

补充：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，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，而线程间可共享内存。

使用多线程的编程通常能够带来更好的性能和用户体验，但多线程程序对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，因为线程间的调度和切换也会浪费CPU时间。时下很时髦的Node.js就采用了单线程异步I/O的工作模式。

58、线程的sleep()和yield()方法有什么区别？

答：

① sleep()给其他线程运行机会时不考虑线程优先级，因此会给低优先级线程运行的机会；

yield()方法只会给相同或更高优先级的线程运行的机会；

② 线程执行sleep()方法后转入阻塞（blocked）状态，

执行yield()方法后转入就绪（ready）状态；

③ sleep()方法声明抛出InterruptedException，而yield()没有声明任何异常；

④ sleep()比yield()（跟操作系统CPU调度相关）有更好的可移植性。

59、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。

60、请说出与线程同步以及线程调度相关的方法。

答：

- wait()：使线程处于等待（阻塞）状态，且释放所持有的对象的锁；

- sleep()：使正在运行的线程处于睡眠状态，静态方法，调用时要处理InterruptedException异常；

- notify()：唤醒一个处于等待状态的线程，在调用此方法时，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，且与优先级无关；

- notityAll()：唤醒所有处于等待状态的线程，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，《关于Java并发编程的总结和思考》。

补充：Java 5通过Lock接口提供了显式的锁机制，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。在对资源访问前，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。

* 没有使用同步机制

•银行账户类：

|  |
| --- |
| /\*\*  \* 银行账户  \*/  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());  }  } |

未同步时，执行结果通常是显示账户余额在10元以下，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  \*/  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  \*/  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  \*/  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;  }  } |

按上述三种方式对代码修改后，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。

61、编写多线程程序有几种实现方式？

答：Java 5以前有两种实现方法：

* 继承Thread类；
* 实现Runnable接口。

都要重写run()方法来定义线程行为，推荐用后者，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后创建线程还有第三种方式：

* 实现Callable接口，接口中call方法可在线程执行结束时产生一个返回值

|  |
| --- |
| 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;  }  @Override  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);  }  } |

62、synchronized关键字的用法？

答：synchronized关键字可将对象或方法标记为同步，以实现对对象和方法的互斥访问，

可用synchronized(对象) { … }定义同步代码块，

或在声明方法时将synchronized作为方法的修饰符。

63、举例说明同步和异步。

答：如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，且不希望让程序等待方法的返回时，就应使用异步编程，很多情况下采用异步往往更有效率。

事实上，同步就是指阻塞式操作，而异步就是非阻塞式操作。

64、启动一个线程是调用run()还是start()方法？

答：启动线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，意味着它可由JVM 调度并执行，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。

65、线程池（thread pool）

答：在面向对象编程中，创建和销毁对象很费时间

因创建一个对象要获取内存资源或其它更多资源。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，需要时从池中获取线程不用自行创建，使用完毕不需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，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的线程，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。

66、线程的基本状态及状态间的关系

答：

* Running表运行状态，
* Runnable表就绪状态（万事俱备，只欠CPU），
* Blocked表阻塞状态:
* 调用wait()方法进入等待池
* 执行同步方法或同步代码块进入等锁池
* 或是调用了sleep()或join()方法等待休眠或其他线程结束
* 或是因发生了I/O中断



67、synchronized 和java.util.concurrent.locks.Lock的异同

答：

相同点：Lock 能完成synchronized所实现的所有功能；

不同点：Lock有比synchronized更精确的线程语义和更好的性能，且不强制性的要求一定要获得锁。

synchronized会自动释放锁，而Lock要求程序员手工释放，且最好在finally 块中释放。

68、序列化

* Java中如何实现序列化，有什么意义？

答：对象流就是将对象的内容进行流化。

可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（如不进行序列化可能会数据乱序）。

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对象输出流，并通过writeObject(Object)方法就可将实现对象写出（即保存其状态）；

如需反序列化，可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

69、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。

编程实现文件拷贝。

|  |
| --- |
| 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。

70、$$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。

答：代码如下：

|  |
| --- |
| 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;  }  } |

Java中字符串中子串的查找:

|  |
| --- |
| 1、int indexOf(String str) ：返回第一次出现的指定子字符串在此字符串中的索引。  2、int indexOf(String str, int startIndex)：从指定的索引处开始，返回第一次出现的指定子字符串在此字符串中的索引。  3、int lastIndexOf(String str) ：返回在此字符串中最右边出现的指定子字符串的索引。  4、int lastIndexOf(String str, int startIndex) ：从指定的索引处开始向后搜索，返回在此字符串中最后一次出现的指定子字符串的索引。 |

例子

|  |
| --- |
| public class Test {  public static void main(String args[]) {  String Str = new String("菜鸟教程:www.runoob.com");  String SubStr1 = new String("runoob");  String SubStr2 = new String("com");  System.out.print("查找字符 o 第一次出现的位置 :" );  System.out.println(Str.indexOf( 'o' ));  System.out.print("从第14个位置查找字符 o 第一次出现的位置 :" );  System.out.println(Str.indexOf( 'o', 14 ));  System.out.print("子字符串 SubStr1 第一次出现的位置:" );  System.out.println( Str.indexOf( SubStr1 ));  System.out.print("从第十五个位置开始搜索子字符串 SubStr1 第一次出现位置 :" );  System.out.println( Str.indexOf( SubStr1, 15 ));  System.out.print("子字符串 SubStr2 第一次出现的位置 :" );  System.out.println(Str.indexOf( SubStr2 ));  }  } |

结果:

|  |
| --- |
| 查找字符 o 第一次出现的位置 :12  从第14个位置查找字符 o 第一次出现的位置 :17  子字符串 SubStr1 第一次出现的位置:9  从第十五个位置开始搜索子字符串 SubStr1 第一次出现的位置 :-1  子字符串 SubStr2 第一次出现的位置 :16 |

71、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;  }  });  }  } |

72、$$ 用Java的套接字编程实现一个多线程的回显（echo）服务器。

答：

|  |
| --- |
| 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);  }  } |

•

73【##】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。

74 [##] 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。

75、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。

76、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。

77、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。

78、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、Proxool、DBCP、BoneCP、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。

79、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。

80、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。

81、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。

82、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();

}

}

}

}

83、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。

84、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》。

85、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")

86、如何通过反射创建对象？

答：

- 方法1：通过类对象调用newInstance()方法，例如：String.class.newInstance()

- 方法2：通过类对象的getConstructor()或getDeclaredConstructor()方法获得构造器（Constructor）对象并调用其newInstance()方法创建对象，例如：String.class.getConstructor(String.class).newInstance("Hello");

87、如何通过反射获取和设置对象私有字段的值？

答：可以通过类对象的getDeclaredField()方法字段（Field）对象，然后再通过字段对象的setAccessible(true)将其设置为可以访问，接下来就可以通过get/set方法来获取/设置字段的值了。下面的代码实现了一个反射的工具类，其中的两个静态方法分别用于获取和设置私有字段的值，字段可以是基本类型也可以是对象类型且支持多级对象操作，例如ReflectionUtil.get(dog, "owner.car.engine.id");可以获得dog对象的主人的汽车的引擎的ID号。

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import java.lang.reflect.Modifier;

import java.util.ArrayList;

import java.util.List;

/\*\*

\* 反射工具类

\*

\*/

public class ReflectionUtil {

private ReflectionUtil() {

throw new AssertionError();

}

/\*\*

\* 通过反射取对象指定字段(属性)的值

\* @param target 目标对象

\* @param fieldName 字段的名字

\* @throws 如果取不到对象指定字段的值则抛出异常

\* @return 字段的值

\*/

public static Object getValue(Object target, String fieldName) {

Class<?> clazz = target.getClass();

String[] fs = fieldName.split("\\.");

try {

for(int i = 0; i < fs.length - 1; i++) {

Field f = clazz.getDeclaredField(fs[i]);

f.setAccessible(true);

target = f.get(target);

clazz = target.getClass();

}

Field f = clazz.getDeclaredField(fs[fs.length - 1]);

f.setAccessible(true);

return f.get(target);

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

/\*\*

\* 通过反射给对象的指定字段赋值

\* @param target 目标对象

\* @param fieldName 字段的名称

\* @param value 值

\*/

public static void setValue(Object target, String fieldName, Object value) {

Class<?> clazz = target.getClass();

String[] fs = fieldName.split("\\.");

try {

for(int i = 0; i < fs.length - 1; i++) {

Field f = clazz.getDeclaredField(fs[i]);

f.setAccessible(true);

Object val = f.get(target);

if(val == null) {

Constructor<?> c = f.getType().getDeclaredConstructor();

c.setAccessible(true);

val = c.newInstance();

f.set(target, val);

}

target = val;

clazz = target.getClass();

}

Field f = clazz.getDeclaredField(fs[fs.length - 1]);

f.setAccessible(true);

f.set(target, value);

}

catch (Exception e) {

throw new RuntimeException(e);

}

}

}

•

88、如何通过反射调用对象的方法？

答：请看下面的代码：

import java.lang.reflect.Method;

class MethodInvokeTest {

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

String str = "hello";

Method m = str.getClass().getMethod("toUpperCase");

System.out.println(m.invoke(str)); // HELLO

}

}

89、简述一下面向对象的"六原则一法则"。

答：

- 单一职责原则：一个类只做它该做的事情。（单一职责原则想表达的就是"高内聚"，写代码最终极的原则只有六个字"高内聚、低耦合"，就如同葵花宝典或辟邪剑谱的中心思想就八个字"欲练此功必先自宫"，所谓的高内聚就是一个代码模块只完成一项功能，在面向对象中，如果只让一个类完成它该做的事，而不涉及与它无关的领域就是践行了高内聚的原则，这个类就只有单一职责。我们都知道一句话叫"因为专注，所以专业"，一个对象如果承担太多的职责，那么注定它什么都做不好。这个世界上任何好的东西都有两个特征，一个是功能单一，好的相机绝对不是电视购物里面卖的那种一个机器有一百多种功能的，它基本上只能照相；另一个是模块化，好的自行车是组装车，从减震叉、刹车到变速器，所有的部件都是可以拆卸和重新组装的，好的乒乓球拍也不是成品拍，一定是底板和胶皮可以拆分和自行组装的，一个好的软件系统，它里面的每个功能模块也应该是可以轻易的拿到其他系统中使用的，这样才能实现软件复用的目标。）

- 开闭原则：软件实体应当对扩展开放，对修改关闭。（在理想的状态下，当我们需要为一个软件系统增加新功能时，只需要从原来的系统派生出一些新类就可以，不需要修改原来的任何一行代码。要做到开闭有两个要点：①抽象是关键，一个系统中如果没有抽象类或接口系统就没有扩展点；②封装可变性，将系统中的各种可变因素封装到一个继承结构中，如果多个可变因素混杂在一起，系统将变得复杂而换乱，如果不清楚如何封装可变性，可以参考《设计模式精解》一书中对桥梁模式的讲解的章节。）

- 依赖倒转原则：面向接口编程。（该原则说得直白和具体一些就是声明方法的参数类型、方法的返回类型、变量的引用类型时，尽可能使用抽象类型而不用具体类型，因为抽象类型可以被它的任何一个子类型所替代，请参考下面的里氏替换原则。）

里氏替换原则：任何时候都可以用子类型替换掉父类型。（关于里氏替换原则的描述，Barbara Liskov女士的描述比这个要复杂得多，但简单的说就是能用父类型的地方就一定能使用子类型。里氏替换原则可以检查继承关系是否合理，如果一个继承关系违背了里氏替换原则，那么这个继承关系一定是错误的，需要对代码进行重构。例如让猫继承狗，或者狗继承猫，又或者让正方形继承长方形都是错误的继承关系，因为你很容易找到违反里氏替换原则的场景。需要注意的是：子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。）

- 接口隔离原则：接口要小而专，绝不能大而全。（臃肿的接口是对接口的污染，既然接口表示能力，那么一个接口只应该描述一种能力，接口也应该是高度内聚的。例如，琴棋书画就应该分别设计为四个接口，而不应设计成一个接口中的四个方法，因为如果设计成一个接口中的四个方法，那么这个接口很难用，毕竟琴棋书画四样都精通的人还是少数，而如果设计成四个接口，会几项就实现几个接口，这样的话每个接口被复用的可能性是很高的。Java中的接口代表能力、代表约定、代表角色，能否正确的使用接口一定是编程水平高低的重要标识。）

- 合成聚合复用原则：优先使用聚合或合成关系复用代码。（通过继承来复用代码是面向对象程序设计中被滥用得最多的东西，因为所有的教科书都无一例外的对继承进行了鼓吹从而误导了初学者，类与类之间简单的说有三种关系，Is-A关系、Has-A关系、Use-A关系，分别代表继承、关联和依赖。其中，关联关系根据其关联的强度又可以进一步划分为关联、聚合和合成，但说白了都是Has-A关系，合成聚合复用原则想表达的是优先考虑Has-A关系而不是Is-A关系复用代码，原因嘛可以自己从百度上找到一万个理由，需要说明的是，即使在Java的API中也有不少滥用继承的例子，例如Properties类继承了Hashtable类，Stack类继承了Vector类，这些继承明显就是错误的，更好的做法是在Properties类中放置一个Hashtable类型的成员并且将其键和值都设置为字符串来存储数据，而Stack类的设计也应该是在Stack类中放一个Vector对象来存储数据。记住：任何时候都不要继承工具类，工具是可以拥有并可以使用的，而不是拿来继承的。）

- 迪米特法则：迪米特法则又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解。（迪米特法则简单的说就是如何做到"低耦合"，门面模式和调停者模式就是对迪米特法则的践行。对于门面模式可以举一个简单的例子，你去一家公司洽谈业务，你不需要了解这个公司内部是如何运作的，你甚至可以对这个公司一无所知，去的时候只需要找到公司入口处的前台美女，告诉她们你要做什么，她们会找到合适的人跟你接洽，前台的美女就是公司这个系统的门面。再复杂的系统都可以为用户提供一个简单的门面，Java Web开发中作为前端控制器的Servlet或Filter不就是一个门面吗，浏览器对服务器的运作方式一无所知，但是通过前端控制器就能够根据你的请求得到相应的服务。调停者模式也可以举一个简单的例子来说明，例如一台计算机，CPU、内存、硬盘、显卡、声卡各种设备需要相互配合才能很好的工作，但是如果这些东西都直接连接到一起，计算机的布线将异常复杂，在这种情况下，主板作为一个调停者的身份出现，它将各个设备连接在一起而不需要每个设备之间直接交换数据，这样就减小了系统的耦合度和复杂度，如下图所示。迪米特法则用通俗的话来将就是不要和陌生人打交道，如果真的需要，找一个自己的朋友，让他替你和陌生人打交道。）

90、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。

91、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？

92、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。

93、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：

类图：

时序图：

94、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

\*

\*/

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;

}

}

}

}

}

95、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）

System.out.println(a == b); // false 两个引用没有引用同一对象

System.out.println(a == c); // true a自动拆箱成int类型再和c比较

}

}

上面你说a自动拆箱成int类型,那他的引用是池中的还是堆中的?