面向对象编程（Object Oriented Programming, OOP）

Java 是一个支持并发、基于类和面向对象的计算机编程语言。下面列出了面向对象软件开发的优点：

- 代码开发模块化，更易维护和修改。
- 代码复用。
- 增强代码的可靠性和灵活性。
- 增加代码的可理解性。
- 面向对象编程有很多重要的特性，比如：封装，继承，多态和抽象。

封装

封装给对象提供了隐藏内部特性和行为的能力。对象提供一些能被其他对象访问的方法来改变它内部的数据。在 Java 当中，有 3 种修饰符：public，private 和 protected。每一种修饰符给其他的位于同一个包或者不同包下面对象赋予了不同的访问权限。

下面列出了使用封装的一些好处：

- 通过隐藏对象的属性来保护对象内部的状态。
- 提高了代码的可用性和可维护性，因为对象的行为可以被单独的改变或者是扩展。
- 禁止对象之间的不良交互，提高模块化。

继承

继承给对象提供了从基类获取字段和方法的能力。继承提供了代码的重用行，也可以在不修改类的情况下给现存的类添加新特性。

Java 采用的是单继承制，使用 extends 关键字。通过继承以后，子类就拥有了父类除私有成员以外的所有成员，从而达到代码重用的目的。在继承过程中，可以通过方法的覆盖来实现多态，让子类拥有自己独特的实现方式。

多态

多态是编程语言给不同的底层数据类型做相同的接口展示的一种能力。一个多态类型上的操作可以应用到其他类型的值上面。

“多态”一词按照字面意思来理解为“多种形式，多种状态”。它的本质是，发送消息给某个对象，让该对象自行决定响应何种行为。通过将子类对象引用赋值给超类对象引用来实现动态方法调用。

抽象

抽象是把想法从具体的实例中分离出来的步骤，因此，要根据他们的功能而不是实现细节来创建类。Java 支持创建只暴露接口而不包含方法实现的抽象的类。这种抽象技术的主要目的是把类的行为和实现细节分离开。

抽象和封装的不同点

抽象和封装是互补的概念。一方面，抽象关注对象的行为。另一方面，封装关注对象行为的细节。一般是通过隐藏对象内部状态信息做到封装，因此，**封装可以看成是用来提供抽象的一种策略。**

一、常见的 Java 问题

1. 什么是 Java 虚拟机？为什么 Java 被称作是“平台无关的编程语言”？

Java 虚拟机是一个可以执行 Java 字节码（class 文件）的虚拟机进程。Java 源文件被编译成能被 Java 虚拟机执行的字节码文件。

Java 被设计成允许应用程序可以运行在任意的平台上，而不需要程序员为每一个平台单独重写或者是重新编译。Java 虚拟机让这个变为可能，因为它知道底层硬件平台的指令长度和其他特性。

2. JDK 和 JRE 的区别是什么？

JDK 是 Java 开发工具，是完整的 Java 软件开发包，它不仅提供了 Java 程序运行所需的 JRE，还提供了一系列的编译、运行等工具，如 javac、javaDoc、javaw 等。JRE 只是 Java 程序的运行环境，它最核心的内容就是 JVM(Java 虚拟机) 以及核心类库，它同时也包含了执行 applet 需要的浏览器插件。

3.“static”关键字是什么意思？Java 中是否可以覆盖(override)一个 private 或者 static 的方法？

“static”关键字表明一个成员变量或者是成员方法可以在没有所属的类的实例变量的情况下被访问。Java 中 static 方法不能被覆盖，因为方法覆盖是基于运行时动态绑定的，而 static 方法是编译时静态绑定的。static 方法跟类的任何实例都不相关，所以概念上不适用。

4. 是否可以在 static 环境中访问非 static 变量？

static 变量在 Java 中是属于类的，它在所有的实例中的值是一样的。当类被 Java 虚拟机载入的时候，会对 static 变量进行初始化。如果你的代码尝试不用实例来访问非 static 的变量，编译器会报错，因为这些变量还没有被创建出来，还没有跟任何实例关联上。

5. Java 支持的数据类型有哪些？什么是自动拆装箱（autoboxing）？

Java 语言支持的 8 种基本数据类型是：

byte short int long float double boolean char

自动装箱是 Java 编译器在基本数据类型和对应的对象包装类型之间做的一个转化。比如：把 int 转化成 Integer，double 转化成 Double 等。反之就是自动拆箱(unboxing)。这个自动转换的过程发生在编译阶段。

6. Java 中的方法覆盖(Overriding)和方法重载(Overloading)是什么意思？

Java 中的方法**重载**发生在同一个类里面两个或者是多个方法的方法名相同但是参数不同的情况。与此相对，方法**覆盖**是说子类重新定义了父类的方法。方法覆盖必须有相同的方法名，参数列表和返回类型。覆盖者可能不会限制它所覆盖的方法的访问。

7. Java 中，什么是构造函数？什么是构造函数重载？什么是复制构造函数？

当新对象被创建的时候，构造函数会被调用。每一个类都有构造函数。在程序员没有给类提供构造函数的情况下，Java 编译器会为这个类创建一个默认的构造函数。

Java 中构造函数重载和方法重载很相似。可以为一个类创建多个构造函数。每一个构造函数必须有它自己唯一的参数列表。

Java 不支持像 C++中那样的复制构造函数，这个不同点是因为如果你不自己写构造函数的情况下，Java 不会创建默认的复制构造函数。

8. Java 支持多继承么？

Java 不支持多继承。每个类都只能继承一个类，但是可以实现多个接口。

9. 接口和抽象类的区别是什么？

Java 提供和支持创建抽象类和接口。它们的实现有共同点，不同点在于：

① 接口中隐含的所有方法都是抽象的。而抽象类则可以同时包含抽象和非抽象的方法。

② 类可以实现很多个接口，但是只能继承一个抽象类。

③ 类如果要实现一个接口，它必须要实现该接口声明的所有方法。但是，类可以不实现抽象类声明的所有方法，当然，在这种情况下，类也必须得声明成是抽象的。

④ 抽象类可以在不提供接口方法实现的情况下实现接口。

⑤ Java 接口中声明的变量默认都是 final 的。抽象类可以包含非 final 的变量。

⑥ Java 接口中的成员函数默认是 public 的。抽象类的成员函数可以是 private，protected 或者是 public。

⑦ 接口是绝对抽象的，不可以被实例化。抽象类也不可以被实例化，但是，如果它包含 main 方法的话是可以被调用的。

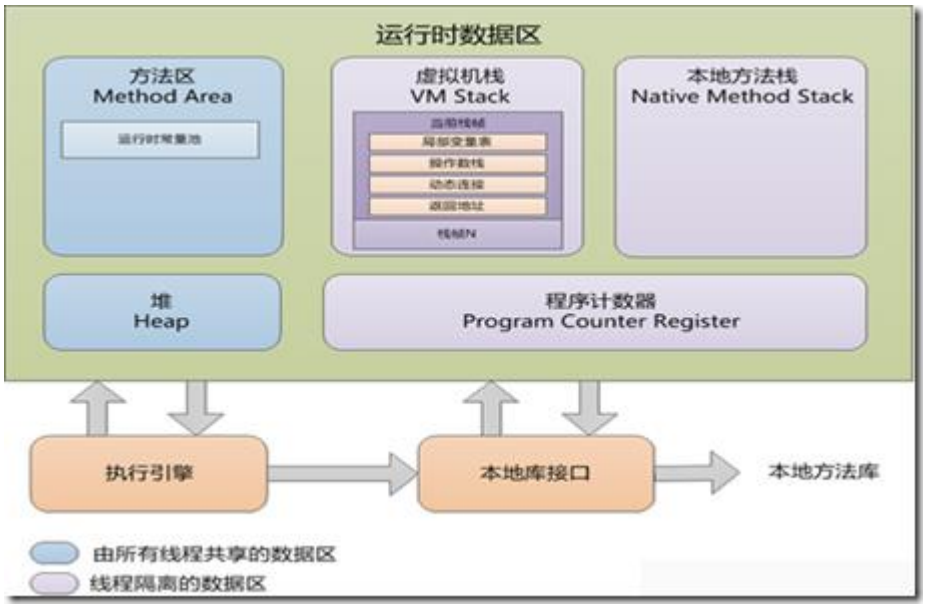
（也可以参考 JDK8 中抽象类和接口的区别）

10. 什么是值传递和引用传址？

对象被值传递，意味着传递了对象的一个副本。因此，就算是改变了对象副本，也不会影响源对象的值。对象被引用传递，意味着传递的并不是实际的对象，而是对象的引用。因此，外部对引用对象所做的改变会反映到所有的对象上。

11. 什么是 JVM 及其工作原理

JVM 是一种用软件模拟出来的计算机，它用于执行 Java 程序，有一套非常严格的技术规范，是 Java 跨平台特性的依赖基础。Java 虚拟机有自己想象中的硬件，如**处理器、堆栈、寄存器**等，还具有相应的指令系统，它运行 Java 程序就好像一台计算机运行 C 或 C++程序一样。Java 虚拟机所管理的内存包括以下几个运行时数据区域：



12. 如何利用 JDK 编译和运行应用程序？

利用 JDK 提供的 javac 命令来编译源文件，利用 java 命令来运行 Java 程序。为了更加方便地利用这两个命令，需要把<JDK 的安装目录>/bin 配置到 Path 环境变量中。

13. 环境变量 CLASSPATH 的作用是什么？

CLASSPATH 环境变量保存的是一些目录和 jar 文件的地址，这些路径是为 Java 程序在编译和运行的时候搜索类而用的，也就是为 Java 程序所依赖的接口、类等指定一个搜索路径。

14. 变量及其作用范围

在 Java 中，变量根据生成周期的不同可以分成**静态变量、成员变量**和**局部变量** 3 种。

静态变量指的是在类中使用 static 修饰的变量，它的生存周期是有类来决定的，当类加载时，它们就生成并初始化。

成员变量则是在类中没有使用 static 修饰的变量，它属于该类的某个实例，也就是对象，它们随着对象的加载而生成并初始化，随着对象被垃圾回收器回收而消失。

局部变量则是定义在方法中的变量、方法的参数或代码块里定义的变量，它们的作用范围用大括号{}来界定，它们随着方法的调用而创建，随着方法的执行完毕而消失。

15. 请简述 Java 中的 main()方法

与 C/C++程序类似，Java 应用程序也需要一个入口函数，它就是 main()方法。main()方法是一个公开的、静态的、无返回值的、参数为一个字符串数组的方法，而且方法名必须为 main，参数 args 与执行参数一一对应。另外，因为 Java 的方法都必须定义在类中，所以 main()方法是属于某一个类的静态方法。

16. Java 中的注释有哪些？

如果不算入 Annotation，Java 的注释有 3 种，即**行注释、块注释和文档注释**。它们往往适合于不同的地方的注释，其中文档注释比较特殊，它的注释信息可以进入 javadoc 文档中。

但是如果把 Annotation 也算作 Java 的注释的话，Java 就有 4 中注释。Annotation 与其他 3 种注释本质的区别在于**它会进入到编译层**，并对程序结果产生影响。它最普遍的作用是用来替代 XML 提供一些配置信息，例如 JPA、Spring 等框架的配置信息就可以通过 Annotation 来提供。

17. 类和对象有什么区别？

Java 的类通过 class 关键字进行定义，它代表了一种抽象的集合，例如，学生类、动物类等，在它的里面可以定义各种属性和方法，它们代表了每个实例的特定的数据和动作。Java 虚拟机对类只加载一次，对它的静态成员也只加载一次。

对象，指的是某一个特定抽象的实例，它属于某一种类型，也就是对象是通过类来创建的。它必须从属于某一个类，通过 new 关键字进行创建，它代表一个特定类型的实例，对象可以可以被多次创建。

简而言之，类是一种抽象，而对象是类的实现。

18. 请介绍 Java 中静态成员的特点

类的静态成员是通过 static 关键字修饰的成员，主要包括：静态成员变量、静态方法和静态代码块，它们具有以下一些特点：

- 在类加载时就进行创建和初始化或执行代码。
- 它们对于一个类来说，都只有一份。
- 类的所有实例都可以访问到它们。

19. 简述 Java 派生类中的构造方法如何为父类传递参数

在 java 中，使用 super()关键字加括号()的形式来为父类的构造方法提供参数，通过参数的数目和类型来决定调用哪一个构造方法。如果调用的是父类的默认的无参数构造方法，则可以不必显式地使用 super()。

20. ″==″和 equals 方法究竟有什么区别？

[分析] == 操作符专门用来比较两个变量的值是否相等，也就是用于比较变量所对应的内存中所存储的数值是否相同，要比较两个基本类型的数据或两个引用变量是否相等，只能用 == 操作符。

如果一个变量指向的数据是对象类型的，这时候就涉及了两块内存，对象本身占用一块内存（堆内存），变量也占用一块内存，例如 Objet obj = new Object(); 变量 obj 是一个内存，new Object()是另一个内存，此时，变量 obj 所对应的内存中存储的数值就是对象占用的那块内存的首地址。对于指向对象类型的变量，如果要比较两个变量是否指向同一个对象，即要看这两个变量所对应的内存中的数值是否相等，这时候就需要用==操作符进行比较。

equals 方法是用于比较两个独立对象的内容是否相同，就好比去比较两个人的长相是否相同，它比较的两个对象是独立的。例如，对于下面的代码：

```
String a=new String("foo");
String b=new String("foo");
```

两条 new 语句创建了两个对象，然后用 a/b 这两个变量分别指向了其中一个对象，这是两个不同的对象，它们的首地址是不同的，即 a 和 b 中存储的数值是不相同的，所以，

表达式 a == b 将返回 false，而这两个对象中的内容是相同的，所以，表达式 a.equals(b)将返回 true。

在实际开发中，我们经常要比较传递进行来的字符串内容是否相等，例如，

String input= …;

input.equals(“quit”);

许多人稍不注意就使用 == 进行比较了，这是错误的，**记住**：字符串的比较基本上都是使用 equals 方法。

如果一个类没有自己定义 equals 方法，那么它将继承 Object 类的 equals 方法，Object 类的 equals 方法的实现代码如下：

```
boolean equals(Object o){
    return this == o;
}
```

这说明，如果一个类没有自己定义 equals 方法，它默认的 equals 方法（从 Object 类继承的）就是使用 == 操作符，也是在比较两个变量指向的对象是否是同一对象，这时候使用 equals 和使用 == 会得到同样的结果，如果比较的是两个独立的对象则总返回 false。如果你编写的类希望能够比较该类创建的两个实例对象的内容是否相同，那么你必须覆盖 equals 方法，由你自己写代码来决定在什么情况即可认为两个对象的内容是相同的。

[答案] equals 和 “==”两者均表示相等的意思，但是它们相等的含义却有所区别。

“==” 运用在基本数据类型时，通过比较它们实际的值来判定是否相同；而用于比较引用类型时，则是比较两个引用地址是否相等，也就是是否指向同一个对象。

equals 方法是 java.lang.Object 的方法，也就是所有的 Java 类都会有的方法。它可以被程序员覆盖重写，通过自定义的方式来判定两个对象是否相等。对于字符串 java.lang.String 类来说，它的 equals 方法用来比较字符串的字符串的字符串序列是否完全相等。

21. String, StringBuffer, StringBuilder 的区别是什么？String 为什么是不可变的？

1、String 是**字符串常量**，StringBuffer 和 StringBuilder 都是**字符串变量**。后两者的字符内容可变，而前者创建后内容不可变。String 对象可以认为是 char 数组的衍生和进一步的封装。它的主要组成部分是：char 数组、偏移量和 string 的长度。

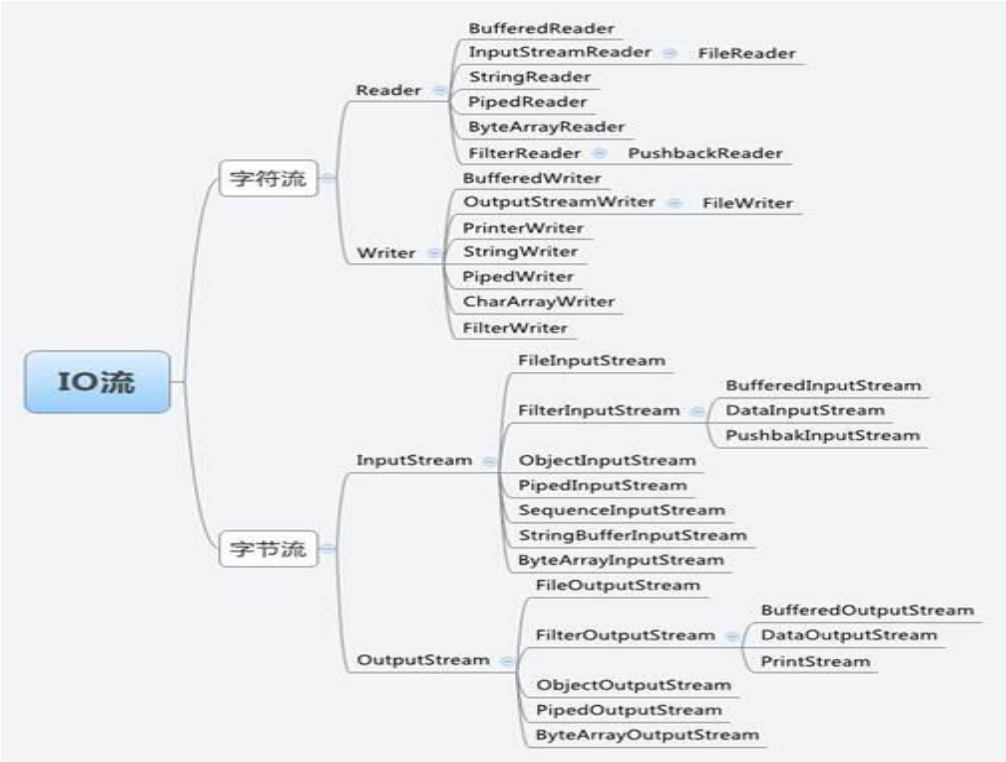
2、String 不可变是因为在 JDK 中 String 类被声明为一个 final 类。不变模式主要作用在当一个对象需要被多线程共享，并且访问频繁时，可以省略同步和锁的等待时间，从而大幅度的提高系统的性能。当两个 string 对象拥有相同的值的时候，他们只引用常量池中同一个拷贝。当同一个字符串反复出现时，可以大幅度的节省内存空间。

3、StringBuffer 是线程安全的，而 StringBuilder 是非线程安全的。

注意： StringBuffer 对几乎所有方法都做了同步，而 StringBuilder 几乎没做任何同步，同步方法需要消耗一定的系统资源，即线程安全会带来额外的系统开销，因此 **StringBuilder 的效率要好于 StringBuffer**。但是在多线程环境下，**StringBuilder 无法保证线程安全**，因此在不考虑线程安全的情况下使用性能较好的 StringBuilder，若系统有线程安全的要求则使用 StringBuffer。

另外，StringBuffer 和 StringBuilder 都可以设置容量大小。

22. 字节流与字符流的区别。



要把一片二进制数据逐一输出到某个设备中，或者从某个设备中逐一读取一片二进制数据，不管输入输出设备是什么，我们要用统一的方式来完成这些操作，用一种抽象的方式进行描述，这个抽象描述方式起名为 IO 流，对应的抽象类为 OutputStream 和 InputStream，不同的实现类就代表不同的输入和输出设备，它们都是针对字节进行操作的。在应用中，经常要完全是字符的一段文本输出去或读进来，用字节流可以吗？计算机中的一切最终都是二

进制的字节形式存在。对于“中国”这些字符，首先要得到其对应的字节，然后将字节写入到输出流。读取时，首先读到的是字节，可是我们要把它显示为字符，我们需要将字节转换成字符。由于这样的需求很广泛，人家专门提供了字符流的包装类。底层设备永远只接受字节数据，有时候要写字符串到底层设备，需要将字符串转成字节再进行写入。字符流是字节流的包装，字符流则是直接接受字符串，它内部将串转成字节，再写入底层设备，这为我们向 IO 设备写入或读取字符串提供了一点点方便。

字符向字节转换时，要注意编码的问题，因为字符串转成字节数组，其实是转成该字符的某种编码的字节形式，读取也是反之的道理。

讲解字节流与字符流关系的代码案例：

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
 import java.io.InputStreamReader;
 import java.io.PrintWriter;
 public class IOTest {
        public static void main(String[]args) throws Exception {
            String str = "中国人";
            /*FileOutputStream fos = new FileOutputStream("1.txt");
            fos.write(str.getBytes("UTF-8"));
            fos.close();*/
            /*FileWriter fw =new FileWriter("1.txt");
            fw.write(str);
            fw.close();*/
            PrintWriter pw =new PrintWriter("1.txt","utf-8");
            pw.write(str);
            pw.close();
            /*FileReader fr =new FileReader("1.txt");
            char[] buf = newchar[1024];
```

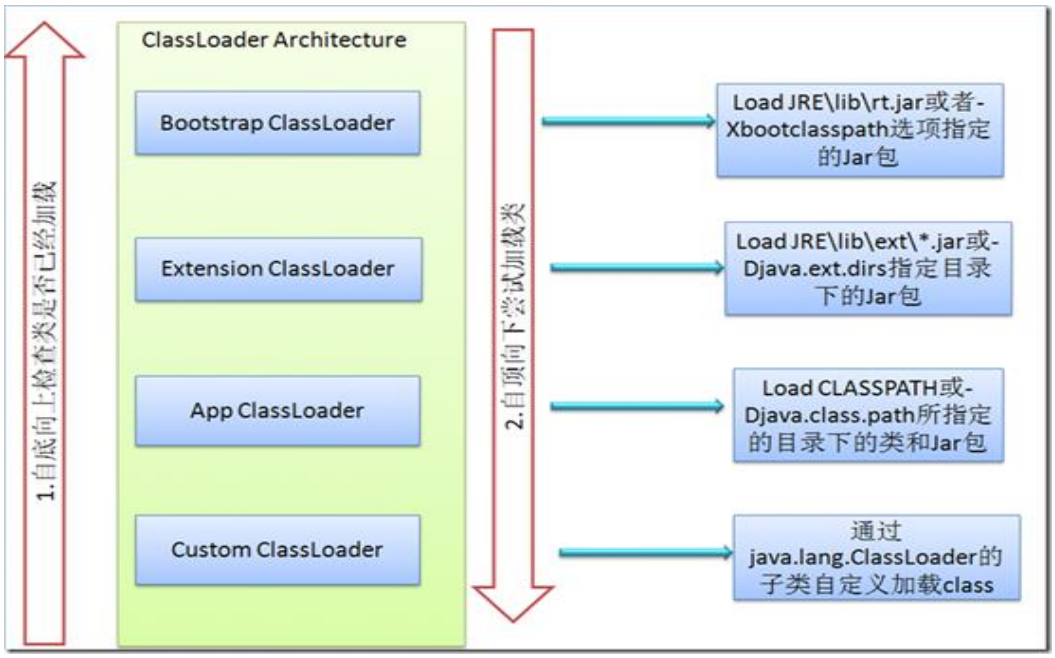
```
            int len =fr.read(buf);
            String myStr = newString(buf,0,len);
            System.out.println(myStr);*/
            /*FileInputStreamfr = new FileInputStream("1.txt");
            byte[] buf = newbyte[1024];
            int len =fr.read(buf);
            String myStr = newString(buf,0,len,"UTF-8");
            System.out.println(myStr);*/
            BufferedReader br =new BufferedReader(
                newInputStreamReader( newFileInputStream("1.txt"),"UTF-8")
            );
        String myStr =br.readLine();
        br.close();
        System.out.println(myStr);
        }
}
```


23. 描述一下 JVM 加载 class 文件的原理机制?

JVM 中类的装载是由 ClassLoader 和它的子类来实现的，Java ClassLoader 是一个重要的 Java 运行时系统组件。它负责在运行时查找和装入类文件的类。

Java 的类加载机制：**ClassLoader**

程序在启动的时候，并不会一次性加载程序所要用的所有 class 文件，而是根据程序的需要，通过 Java 的类加载机制（ClassLoader）来动态加载某个 class 文件到内存当中的，从而只有 class 文件被载入到了内存之后，才能被其它 class 所引用。所以 ClassLoader 就是用来**动态加载 class 文件到内存当中**用的。



24. heap（堆）和 stack（栈）有什么区别？

java 的内存分为两类，一类是**栈内存**，一类是**堆内存**。

栈内存是指程序进入一个方法时，会为这个方法单独分配一块私属存储空间，用于存储这个方法内部的局部变量，当这个方法结束时，分配给这个方法的栈会释放，这个栈中的变量也将随之释放。

堆与栈作用的是不同的内存，一般用于存放在当前方法栈中的那些数据，例如，使用 new 创建的对象都放在堆里，所以它不会随方法的结束而消失。方法中的局部变量使用 final 修饰后，放在堆中，而不是栈中。

25. 关于 JAVA 内存模型，一个对象（两个属性，四个方法）实例化 100 次，现在内存中的存储状态，几个对象，几个属性，几个方法？

由于 JAVA 中 new 出来的对象都是放在堆中，所以如果要实例化 100 次，将在堆中产生 100 个对象，一般对象与其中的属性、方法都属于一个整体，但如果属性和方法是静态的，就是用 static 关键字声明的，那么属于类的属性和方法永远只在内存中存在一份。

二、Java 线程

1. 进程和线程的区别是什么？

进程是执行着的应用程序，而线程是进程内部的一个执行序列。一个进程可以有多个线程。线程又叫做轻量级进程。

2. 创建线程有几种不同的方式？你喜欢哪一种？为什么？

有三种方式可以用来创建线程：

① 继承 Thread 类

② 实现 Runnable 接口

③ 应用程序可以使用 Executor 框架来创建线程池

实现 **Runnable 接口**这种方式更受欢迎，因为这不需要继承 Thread 类。在应用设计中已经继承了别的对象的情况下，这需要多继承（而 Java 不支持多继承），只能实现接口。同时，线程池也是非常高效的，很容易实现和使用。

3. 概括的解释下线程的几种可用状态。

线程在执行过程中，可以处于下面几种状态：

① 就绪(Runnable)：线程准备运行，不一定立马就能开始执行。

② 运行中(Running)：进程正在执行线程的代码。

③ 等待中(Waiting)：线程处于阻塞的状态，等待外部的处理结束。

④ 睡眠中(Sleeping)：线程被强制睡眠。

⑤ I/O 阻塞(Blocked on I/O)：等待 I/O 操作完成。

⑥ 同步阻塞(Blocked on Synchronization)：等待获取锁。

⑦ 死亡(Dead)：线程完成了执行。

4. 同步方法和同步代码块的区别是什么？

在 Java 语言中，每一个对象有一把锁。线程可以使用 synchronized 关键字来获取对象上的锁。synchronized 关键字可应用在方法级别(粗粒度锁)或者是代码块级别(细粒度锁)。

5. 在监视器(Monitor)内部，是如何做线程同步的？程序应该做哪种级别的同步？

监视器和锁在 Java 虚拟机中是一块使用的。监视器监视一块同步代码块，确保一次只有一个线程执行同步代码块。每一个监视器都和一个对象引用相关联。线程在获取锁之前不允许执行同步代码。

6 .什么是死锁(deadlock)？

两个进程都在等待对方执行完毕才能继续往下执行的时候就发生了死锁。结果就是两个进程都陷入了无限的等待中。

7. 如何确保 N 个线程可以访问 N 个资源，同时又不导致死锁？

使用多线程的时候，一种非常简单的避免死锁的方式就是：指定获取锁的顺序，并强制线程按照指定的顺序获取锁。因此，如果所有的线程都是以同样的顺序加锁和释放锁，就不会出现死锁了。

8. sleep()和 wait()有什么区别?

sleep() 是线程类（Thread）的方法，导致此线程暂停执行指定时间，将执行机会让给给其他线程，但是监控状态依然保持，到时后会自动恢复。调用 sleep 不会释放对象锁。

wait() 是 Object 类的方法，对此对象调用 wait()方法导致本线程放弃对象锁，进入等待此对象的等待锁定池，只有针对此对象发出 notify 方法（或 notifyAll）后本线程才进入对象锁定池准备获得对象锁进入运行状态。

sleep() 就是正在执行的线程主动让出 cpu，cpu 去执行其他线程，在 sleep 指定的时间过后，cpu 才会回到这个线程上继续往下执行，如果当前线程进入了同步锁，sleep 方法并不会释放锁，即使当前线程使用 sleep 方法让出了 cpu，但其他被同步锁挡住了的线程也无法得到执行。

wait() 是指在一个已经进入了同步锁的线程内，让自己暂时让出同步锁，以便其他正在等待此锁的线程可以得到同步锁并运行，只有其他线程调用了 notify 方法（notify 并不释放锁，只是告诉调用过 wait 方法的线程可以去参与获得锁的竞争了，但不是马上得到锁，因为锁还在别人手里，别人还没释放。如果 notify 方法后面的代码还有很多，需要这些代码执行完后才会释放锁，可以在 notfiy 方法后增加一个等待和一些代码，看看效果），调用 wait 方法的线程就会解除 wait 状态和程序可以再次得到锁后继续向下运行。对于 wait 的讲解一定要配合例子代码来说明，才显得自己真明白。

```
package com.huawei.interview;
    public class MultiThread {
        public static void main(String[] args) {
            new Thread(newThread1()).start();
            try {
                Thread.sleep(10);
             } catch (InterruptedException e) {
                    e.printStackTrace();
                    }
```

```
                new Thread(newThread2()).start();
            }
            private static class Thread1 implements Runnable {
            @Override
            public void run() {
```

//由于这里的 Thread1 和下面的 Thread2 内部 run 方法要用同一对象作为监视器，我们这里不能用 this，因为在 Thread2 里面的 this 和这个 Thread1 的 this 不是同一个对象。我们用 MultiThread.class 这个字节码对象，当前虚拟机里引用这个变量时，指向的都是同一个对象。

```
        synchronized (MultiThread.class){
            System.out.println("enter thread1...");
            System.out.println("thread1 is waiting");
                try {
```

//释放锁有两种方式，第一种方式是程序自然离开监视器的范围，也就是离开了 synchronized 关键字管辖的代码范围，另一种方式就是在 synchronized 关键字管辖的代码内部调用监视器对象的 wait 方法。这里，使用 wait 方法释放锁。

```
            MultiThread.class.wait();
            } catch(InterruptedException e) {
                    e.printStackTrace();
             }
             System.out.println("thread1is going on...");
             System.out.println("thread1is being over!");
           }
      }
}
        private static class Thread2 implements Runnable{
            @Override
            public void run() {
             synchronized (MultiThread.class){
                System.out.println("enterthread2...");
```

```
            System.out.println("thread2 notify other thread can release wait status..");
        //由于 notify 方法并不释放锁， 即使 thread2 调用下面的 sleep 方法休息了 10 毫秒，但 thread1 仍然不会执行，因为 thread2 没有释放锁，所以 Thread1 无法得不到锁。
                MultiThread.class.notify();
                System.out.println("thread2   is sleeping ten millisecond...");
                try {
                    Thread.sleep(10);
                        e.printStackTrace();
                    }
                    System.out.println("thread2is going on...");
                    System.out.println("thread2is being over!");
                    }
                }
            }
        }
```


9. 同步和异步有何异同，在什么情况下分别使用他们？

举例说明。如果数据将在线程间共享。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，那么这些数据就是共享数据，必须进行同步存取。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。

10. 当一个线程进入一个对象的一个 synchronized 方法后，其它线程是否可进入此对象的其它方法?

分几种情况讨论：

1. 其他方法前面是否加了 synchronized 关键字，如果没加，则能。

2. 如果这个方法内部调用了 wait，则可以进入其他 synchronized 方法。

3. 如果其他个方法都加了 synchronized 关键字，并且内部没有调用 wait，则不能。

4. 如果其他方法是 static，它用的同步锁是当前类的字节码，与非静态的方法不能 同步，因为非静态的方法用的是 this。

11. 线程同步，并发操作怎么控制？

Java 中可在方法名前加**关键字 syschronized** 来处理当有多个线程同时访问共享资源时的问题。syschronized 相当于一把锁，当有申请者申请该资源时，如果该资源没有被占用，那么将资源交付给这个申请者使用，在此期间，其他申请者只能申请而不能使用该资源，当该资源被使用完成后将释放该资源上的锁，其他申请者可申请使用。

并发控制主要是为了多线程操作时带来的资源读写问题。如果不加以控制可能会出现死锁，读脏数据、不可重复读、丢失更新等异常。

并发操作可以通过加锁的方式进行控制，锁又可分为乐观锁和悲观锁。

乐观锁：假定系统的数据修改只会产生非常少的冲突，也就是说任何进程都不大可能修改别的进程正在访问的数据。乐观并发模式下，读数据和写数据之间不会发生冲突，只有写数据与写数据之间会发生冲突。即读数据不会产生阻塞，只有写数据才会产生阻塞。

悲观锁：并发模式假定系统中存在足够多的数据修改操作，以致于任何确定的读操作都可能会受到由个别的用户所制造的数据修改的影响。也就是说悲观锁假定冲突总会发生，通过独占正在被读取的数据来避免冲突。但是独占数据会导致其他进程无法修改该数据，进而产生阻塞，读数据和写数据会相互阻塞。

三、Java 集合类

1. Java 集合类框架的基本接口有哪些？

Java 集合类提供了一套设计良好的支持对一组对象进行操作的接口和类。Java 集合类里最基本的接口有：

Collection

Collection 是最基本的集合接口，一个 Collection 代表一组 Object 的集合，这些 Object 被称作 Collection 的元素。根据用途的不同，Collection 又划分为 List 与 Set。

List（列表）

继承自 Collection 接口的有序集合，可以根据索引（元素在 List 中的位置，类似于数组下标）进行取得/删除/插入操作。并且允许有重复元素，对于满足 e1.equals(e2)条件的 e1 与 e2 对象元素，可以同时存在于 List 集合中。当然，也有 List 的实现类不允许重复元素的存在。除了具有 Collection 接口必备的 iterator()方法外，List 还提供一个 listIterator()方法，返回一个 ListIterator 接口对象，和 Iterator 接口相比，ListIterator 增加了元素的添加，

删除，和设定等方法，还能向前或向后遍历。实现 List 接口的常用类有 LinkedList，ArrayList，Vector 和 Stack。

① LinkedList 类

LinkedList 实现了 List 接口， 允许 null 元素。 此外 LinkedList 提供额外的 get，remove， insert 方法在 LinkedList 的首部或尾部。这些操作使 LinkedList 可被用作堆栈（stack），队列（queue）或双向队列（deque）。

注意 LinkedList 没有同步方法。 如果多个线程同时访问一个 List， 则必须自己实现访问同步。一种解决方法是在创建 List 时构造一个同步的 List：

List list = Collections.synchronized List(new LinkedList(...));

② ArrayList 类

ArrayList 实现了可变大小的数组。它允许所有元素，包括 null。ArrayList 没有同步。size，isEmpty，get，set 方法运行时间为常数。但是 add 方法开销为分摊的常数，添加 n 个元素需要 O(n)的时间。其他的方法运行时间为线性。每个 ArrayList 实例都有一个容量（Capacity），即用于存储元素的数组的大小。这个容量可随着不断添加新元素而自动增加，但是增长算法并没有定义。当需要插入大量元素时，在插入前可以调用 ensureCapacity 方法来增加 ArrayList 的容量以提高插入效率。和 LinkedList 一样，ArrayList 也是非同步的（unsynchronized）。

③ Vector 类

Vector 非常类似 ArrayList， 但是 Vector 是同步的。 由 Vector 创建的 Iterator， 虽然和 ArrayList 创建的 Iterator 是同一接口，但是，因为 Vector 是同步的，当一个 Iterator 被创建而且正在被使用， 另一个线程改变了 Vector 的状态 （例如， 添加或删除了一些元素）， 这时调用 Iterator 的方法时将抛出 ConcurrentModificationException，因此必须捕获该异常。

④Stack 类

Stack 继承自 Vector，实现一个后进先出的堆栈。Stack 提供 5 个额外的方法使得 Vector 得以被当作堆栈使用。基本的 push 和 pop 方法，还有 peek 方法得到栈顶的元素，empty 方法测试堆栈是否为空， search 方法检测一个元素在堆栈中的位置。Stack 刚创建后是空栈。

Set

继承自 Collection 接口的无序集合，不能包含重复元素，可以存放不同类型的对象。对于满足 e1.equals(e2)条件的 e1 与 e2 对象元素，不能同时存在于同一个 Set 集合里，换句话说，Set 集合里任意两个元素 e1 和 e2 都满足 e1.equals(e2)==false 条件，Set 最多有一个 null 元素。因为 Set 的这个制约，在使用 Set 集合时应注意：

1 为 Set 集合里的元素的实现类实现一个有效的 equals(Object)方法。

2 对 Set 的构造函数，传入的 Collection 参数不能包含重复的元素。

Map（映射）

Map **没有继承 Collection 接口**，也就是说 Map 和 Collection 是两种不同的集合。Collection 可以看作是（value）的集合，而 Map 可以看作是（key，value）的集合。Map 接口由 Map 的内容提供 3 种类型的集合视图，一组 key 集合，一组 value 集合，或者一组 key-value 映射关系的集合。Map 中不能包含相同的 key，每个 key 只能映射一个 value。键值对（如手机中的电话本）

① Hashtable 类

java.lang.Object


java.util.Dictionary<K, V>


java.util.Hashtable<K, V>

public class Hashtable<K,V> extends Dictionary<K,V>

implements Map<K,V>, Cloneable, java.io.Serializable { }

Hashtable 继承于 Dictionary 类，实现了 Map 接口，实现一个 key-value 映射的哈希表。任何非空（non-null）的对象都可作为 key 或者 value。添加数据使用 put(key, value)， 取出数据使用 get(key)， 这两个基本操作的时间开销为常数。Hashtable 通过初始容量（initial capacity）和加载因子（load factor）两个参数调整性能。通常默认的 load factor 是 0.75，较好地实现了时间和空间的均衡。增大 load factor 可以节省空间但相应的查找时间将增大，这会影响像 get 和 put 这样的操作。使用 Hashtable 的简单示例如下，将 1，2，3 放到 Hashtable 中，他们的 key 分别是”one”，”two”，”three”：

代码如下:

```
Hashtable numbers = new Hashtable();
numbers.put(“one”, new Integer(1));
```

numbers.put(“two”, new Integer(2));

numbers.put(“three”, new Integer(3));

要取出一个数，比如 2，用相应的 key：

Integer n = (Integer)numbers.get(“two”);

System.out.println(“two = ” + n);

由于作为 key 的对象将通过计算其散列函数来确定与之对应的 value 的位置， 因此任何作为 key 的对象都必须实现 hashCode 和 equals 方法。hashCode 和 equals 方法继承自根类 Object，如果你用自定义的类当作 key 的话，要相当小心，按照散列函数的定义，如果两个对象相同，即 obj1.equals(obj2)=true，则它们的 hashCode 必须相同，但如果两个对象不同，则它们的 hashCode 不一定不同，如果两个不同对象的 hashCode 相同，这种现象称为冲突，冲突会导致操作哈希表的时间开销增大，所以尽量定义好的 hashCode()方法，能加快哈希表的操作。如果相同的对象有不同的 hashCode，对哈希表的操作会出现意想不到的结果（期待的 get 方法返回 null），要避免这种问题，只需要牢记一条：要同时复写 equals 方法和 hashCode 方法，而不要只写其中一个。Hashtable 是同步的。

② HashMap 类

HashMap 和 Hashtable 类似，不同之处在于 HashMap 是非同步的，并且允许 null，即 null value 和 null key。但是将 HashMap 视为 Collection 时（values()方法可返回 Collection），其迭代子操作时间开销和 HashMap 的容量成比例。因此，如果迭代操作的性能相当重要的话，不要将 HashMap 的初始化容量设得过高，或者 load factor 过低。

③ WeakHashMap 类

WeakHashMap 是一种改进的 HashMap，它对 key 实行“弱引用”，如果一个 key 不再被外部所引用，那么该 key 可以被 GC 回收。

［注意］

Java SDK 不提供直接继承自 Collection 的类，Java SDK 提供的类都是继承自 Collection 的“子接口”如 List 和 Set。

所有实现 Collection 接口的类都必须提供两个标准的构造函数： 无参数的构造函数用于创建一个空的 Collection，有一个 Collection 参数的构造函数用于创建一个新的 Collection， 这个新的 Collection 与传入的 Collection 有相同的元素。后一个构造函数允许用户复制一个 Collection。 Collections 是针对集合类的一个帮助类，他提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

如何遍历 Collection 中的每一个元素？

不论 Collection 的实际类型如何，它都支持一个 iterator()的方法，该方法返回一个迭代子，使用该迭代子可逐一访问 Collection 中每一个元素。典型的用法如下：

复制代码如下:

```
Iterator it = collection.iterator();      // 获得一个迭代子
while(it.hasNext()) {
    Object obj = it.next();               // 得到下一个元素
}
```

① 如果涉及到堆栈，队列等操作，应该考虑用 List，对于需要快速插入，删除元素，应该使用 LinkedList，如果需要快速随机访问元素，应该使用 ArrayList。

② 如果程序在单线程环境中， 或者访问仅仅在一个线程中进行， 考虑非同步的类，其效率较高。

③ 如果多个线程可能同时操作一个类，应该使用同步的类。要特别注意对哈希表的操作，作为 key 的对象要正确复写 equals 和 hashCode 方法。

④ 尽量返回接口而非实际的类型，如返回 List 而非 ArrayList，这样如果以后需要将 ArrayList 换成 LinkedList 时，客户端代码不用改变。这就是针对抽象编程。

⑤ HashSet 按照 hashcode 值的某种运算方式进行存储， 而不是直接按 hashCode 值的大小进行存储。例如，"abc"---> 78，"def" ---> 62，"xyz" ---> 65 在 hashSet 中的存储顺序不是 62,65,78。LinkedHashSet 按插入的顺序存储，那被存储对象的 hashcode 方法还有什么作用呢？hashset 集合比较两个对象是否相等，首先看 hashcode 方法是否相等， 然后看 equals 方法是否相等。new 两个 Student 插入到 HashSet 中，看 HashSet 的 size，实现 hashcode 和 equals 方法后再看 size。

同一个对象可以在 Vector 中加入多次。往集合里面加元素，相当于集合里用一根绳子连接到了目标对象。往 HashSet 中却加不了多次的。

2. 为什么集合类没有实现 Cloneable 和 Serializable 接口？

集合类接口指定了一组叫做元素的对象。集合类接口的每一种具体的实现类都可以选择以它自己的方式对元素进行保存和排序。有的集合类允许重复的键，有些不允许。

克隆(cloning)或者是序列化(serialization)的语义和含义是跟具体的实现相关的。因此，应该由集合类的具体实现来决定如何被克隆或者是序列化。

3. 什么是迭代器(Iterator)？

Iterator 接口提供了很多对集合元素进行迭代的方法。每一个集合类都包含了可以返回迭代器实例的迭代方法。迭代器可以在迭代的过程中删除底层集合的元素。

迭代器，提供一种访问一个集合对象中各个元素的途径，同时又不需要暴露该对象的内部细节。Java 通过提供 Iterable 和 Iterator 两个接口来实现集合类的可迭代性。迭代器主要的用法就是首先用 hasNext()作为循环条件；再用 next()方法得到每一个元素；最后再进行相关的操作。（此外还有一个 remove()方法：从迭代器指向的 Collection 中移除迭代器返回的最后一个元素，该操作使用的比较少。）

4. Iterator 和 ListIterator 的区别是什么？

下面列出了他们的区别：

① Iterator 可用来遍历 Set 和 List 集合，但是 ListIterator 只能用来遍历 List。

② Iterator 对集合只能是前向遍历，ListIterator 既可以前向也可以后向。

③ ListIterator 实现了 Iterator 接口，并包含其他的功能，比如：增加元素，替换元素，获取前一个和后一个元素的索引，等等。

④ ListIterator 有 add()方法，可以向 List 中添加对象，而 Iterator 不能。

⑤ ListIterator 和 Iterator 都有 hasNext()和 next()方法，可以实现顺序向后遍历，但是 ListIterator 有 hasPrevious()和 previous()方法，可以实现逆向（顺序向前）遍历。Iterator 就不可以。

⑥ListIterator 可以定位当前的索引位置，nextIndex()和 previousIndex()可以实现。Iterator 没有此功能。

⑦都可实现删除对象，但是 ListIterator 可以实现对象的修改，set()方法可以实现。Iterator 仅能遍历，不能修改。

5. 快速失败(fail-fast)和安全失败(fail-safe)的区别是什么？

Iterator 的安全失败是基于对底层集合做拷贝，因此，它不受源集合上修改的影响。

java.util 包下面的所有的集合类都是快速失败的，而 java.util.concurrent 包下面的所有的类都是安全失败的。

快速失败的迭代器会抛出 ConcurrentModificationException 异常，而安全失败的迭代器永远不会抛出这样的异常。

6. Java 中的 HashMap 的工作原理是什么？

Java 中的 HashMap 是以键值对(key-value)的形式存储元素的。HashMap 需要一个 hash 函数，它使用 hashCode()和 equals()方法来向集合/从集合添加和检索元素。当调用 put()方法的时候，HashMap 会计算 key 的 hash 值，然后把键值对存储在集合中合适的索引上。如果 key 已经存在了，value 会被更新成新值。HashMap 的一些重要的特性是它的容量(capacity)，负载因子(load factor)和扩容极限(threshold resizing)。

7. hashCode()和 equals()方法的重要性体现在什么地方？

Java 中的 HashMap 使用 hashCode()和 equals()方法来确定键值对的索引，当根据键获取值的时候也会用到这两个方法。如果没有正确地实现这两个方法，两个不同的键可能会有相同的 hash 值，因此，可能会被集合认为是相等的。而且，这两个方法也用来发现重复元素。所以这两个方法的实现对 HashMap 的精确性和正确性是至关重要的。

8. HashMap 和 HashTable 有什么区别？

HashMap 是 HashTable 的轻量级实现（非线程安全的实现），他们都继承自 Map 接口，HashTable 继承自 Dictionary 类， 而 HashMap 是 Java1.2 引进的 Map interface 的一个实现。

HashTable 和 HashMap 采用的 hash/rehash 算法都大概一样，所以性能不会有很大的差异。但是，它们有以下**不同点**：

①HashTable 不允许键或者值是 null，HashMap 允许键和值是 null，由于非线程安全，在只有一个线程访问的情况下，效率要高于 HashTable。

② HashTable 的方法是允许同步的，而 HashMap 不是。因此，HashMap 更适合于单线程环境，而 HashTable 适合于多线程环境。在多个线程访问 HashTable 时，不需要自己为它的方法实现同步，而 HashMap 就必须为之提供外同步。

③ HashTable 有一个 contains()方法，功能和 containsValue()功能一样。HashMap 把 HashTable 的 contains 方法去掉了，改成 containsValue 和 containsKey。因为 contains 方法容易让人引起误解。

④ HashTable 使用 Enumeration，HashMap 使用 Iterator。

⑤ hash 数组的初始化大小及增长方式不同。HashTable 中 hash 数组的默认大小是 11，增加方式的 old*2+1，HashMap 中 hash 数组的默认大小是 16，增长方式一定是 2 的指数倍。

⑥ 哈希值的使用不同，HashTable 直接使用对象的 hashCode，而 HashMap 会重新计算 hash 值。

⑦ HashMap 提供了可供应用迭代的键的集合，因此，HashMap 是快速失败的。另一方面，HashTable 提供了对键的列举(Enumeration)。一般认为 HashTable 是一个遗留的类。

9. 数组(Array)和列表(ArrayList)有什么区别？什么时候应该使用 Array 而不是 ArrayList？

下面列出了 Array 和 ArrayList 的**不同点**：

①Array 可以包含基本类型和对象类型，ArrayList 只能包含对象类型。

②Array 大小是固定的，ArrayList 的大小是动态变化的。

ArrayList 提供了更多的方法和特性，比如：addAll()，removeAll()，iterator()等

对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢。

10. ArrayList 和 LinkedList 有什么区别？

ArrayList 和 LinkedList 都实现了 List 接口，它们有以下的**不同点**：

①ArrayList 是基于索引的数据接口，它的底层是数组。它可以以 O(1)时间复杂度对元素进行随机访问。

②与此对应，LinkedList 是以元素列表（链表）的形式存储它的数据，每一个元素都和它的前一个和后一个元素链接在一起，在这种情况下，查找某个元素的时间复杂度是 O(n)。

③相对于 ArrayList，LinkedList 的插入、添加、删除操作速度更快，因为当元素被添加到集合任意位置的时候，不需要像数组那样重新计算大小或者是更新索引。

④LinkedList 比 ArrayList 更占内存，因为 LinkedList 为每一个节点存储了两个引用，一个指向前一个元素，一个指向下一个元素。

（可以参考 ArrayList vs. LinkedList）

11. Comparable 和 Comparator 接口是干什么的？列出它们的区别。

Java 提供了只包含一个 compareTo()方法的 Comparable 接口。这个方法可以给两个对象排序。具体来说，它返回负数，0，正数来表明输入对象小于，等于，大于已经存在的对象。

Java 提供了包含 compare()和 equals()两个方法的 Comparator 接口（比较器）。compare()方法用来给两个输入参数排序，返回负数，0，正数表明第一个参数是小于，等于，大于第二个参数。equals()方法需要一个对象作为参数，它用来决定输入参数是否和 comparator 相等。只有当输入参数也是一个 comparator 并且输入参数和当前 comparator 的排序结果是相同的时候，这个方法才返回 true。

比较器是把集合或数组的元素强行按照指定方法进行排序的对象，它是实现了Comparator接口类的实例。如果一个集合元素的类型是可比较的(实现了Comparable接口)，那么它就具有了默认的排序方法，比较器则是强行改变它默认的比较方式来进行排序。或者有的集合元素不可比较（没有实现 Comparable 接口），则可用比较器来实现动态的排序。

12. 什么是 Java 优先级队列(Priority Queue)？

PriorityQueue 是一个基于优先级堆的无界队列，它的元素是按照自然顺序(natural order)排序的。在创建的时候，我们可以给它提供一个负责给元素排序的比较器。PriorityQueue 不允许 null 值，因为他们没有自然顺序，或者说他们没有任何的相关联的比较器。最后，PriorityQueue 不是线程安全的，入队和出队的时间复杂度是 O(log(n))。

13. 你了解大 0 符号(big-0 notation)么？你能给出不同数据结构的例子么？

O 符号描述了当数据结构里面的元素增加的时候，算法的规模或者是性能在最坏的场景下有多么好。

大 O 符号也可用来描述其他的行为，比如：内存消耗。因为集合类实际上是数据结构，我们一般使用大 O 符号基于时间，内存和性能来选择最好的实现。大 O 符号可以对大量数据的性能给出一个很好的说明。

14. 如何权衡是使用无序的数组还是有序的数组？

有序数组最大的好处在于查找的时间复杂度是 O(log n)，而无序数组是 O(n)。

有序数组的缺点是插入操作的时间复杂度是 O(n)，因为值大的元素需要往后移动来给新元素腾位置。相反，无序数组的插入时间复杂度是常量 O(1)。

15. Java 集合类框架的最佳实践有哪些？

根据应用的需要正确选择要使用的集合的类型对性能非常重要，比如：

假如元素的大小是固定的， 而且能事先知道，我们就应该用 Array 而不是 ArrayList。

有些集合类允许指定初始容量。因此，如果我们能估计出存储的元素的数目，我们可以设置初始容量来避免重新计算 hash 值或者是扩容。

为了类型安全，可读性和健壮性的原因总是要使用泛型。同时，使用泛型还可以避免运行时的 ClassCastException。

使用 JDK 提供的不变类(immutable class)作为 Map 的键可以避免为我们自己的类实现 hashCode()和 equals()方法。

编程的时候接口优于实现。

底层的集合实际上是空的情况下，返回长度是 0 的集合或者是数组，不要返回 null。

16. Enumeration 接口和 Iterator 接口的区别有哪些？

Enumeration 速度是 Iterator 的 2 倍，同时占用更少的内存。但是，Iterator 远远比 Enumeration 安全，因为其他线程不能够修改正在被 iterator 遍历的集合里面的对象。

同时，Iterator 允许调用者删除底层集合里面的元素，这对 Enumeration 来说是不可能的。

17. HashSet 和 TreeSet 有什么区别？

HashSet 是由一个 hash 表来实现的，因此，它的元素是无序的。add()，remove()，contains()方法的时间复杂度是 O(1)。

另一方面，TreeSet 是由一个树形的结构来实现的，它里面的元素是有序的。因此，add()，remove()，contains()方法的时间复杂度是 O(logn)。

18. Vector 与 ArrayList 的区别

这两个类都实现了 List 接口，且都是有序集合，即存储在这两个集合中的元素的位置都是有顺序的，相当于一种动态的数组，可以按位置索引号取出某个元素，并且其中的数据是允许重复的，这是与 HashSet 之类的集合的最大不同处，HashSet 之类的集合不可以按索引号去检索其中的元素，也不允许有重复的元素。

ArrayList 与 Vector 的区别，这主要包括**两个方面**：.

（1）同步性：

Vector 是线程安全的，也就是说它的方法之间是线程同步的，而 ArrayList 是线程不安全的，它的方法之间是线程不同步的。如果只有一个线程会访问到集合，那最好是使用 ArrayList，因为它不考虑线程安全，效率会高些；如果有多个线程会访问到集合，那最好是使用 Vector，因为不需要我们自己再去考虑和编写线程安全的代码。

（2）数据增长：

ArrayList 与 Vector 都有一个初始的容量大小，当存储进它们里面的元素的个数超过了容量时，就需要增加 ArrayList 与 Vector 的存储空间，每次要增加存储空间时，不是只增加一个存储单元，而是增加多个存储单元，每次增加的存储单元的个数在内存空间利用与程序效率之间要取得一定的平衡。Vector 默认增长为原来的两倍，而 ArrayList 的增长策略在文档中没有明确规定（从源代码看到的是增长为原来的 1.5 倍）。ArrayList 与 Vector 都可以设置初始的空间大小，Vector 还可以设置增长的空间大小，而 ArrayList 没有提供设置增长空间的方法。

备注：对于 Vector&ArrayList、Hashtable&HashMap，要记住线程安全的问题，记住 Vector 与 Hashtable 是旧的， 是 java 一诞生就提供了的， 它们是线程安全的，

ArrayList 与 HashMap 是 java2 时才提供的，它们是线程不安全的。

19. List 和 Map 区别?

一个是存储单列数据的集合，另一个是存储键和值这样的双列数据的集合，List 中存储的数据是有顺序，并且允许重复；Map 中存储的数据是没有顺序的，其键是不能重复的，它的值是可以有重复的。

20. Collection 和 Collections 的区别?

Collections 是个 java.util 下的类，Collections 是针对集合类的一个帮助类，他提供一系列有关集合操作的静态方法，实现对各种集合的搜索、排序、线程安全化等操作。Collection 是个 java.util 下的接口，它是各种集合结构的父接口。

21. 说出 ArrayList,Vector, LinkedList 的存储性能和特性。

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 由于使用了 synchronized 方法（线程安全），通常性能上较 ArrayList 差，而 LinkedList 使用双向链表实现存储，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。LinkedList 也是线程不安全的，LinkedList 提供了一些方法，使得 LinkedList 可以被当作堆栈和队列来使用。

22. 集合使用泛型带来的了什么好处？

使用泛型后，可以达到元素类型明确的目的，避免了手动类型转换的过程，同时，也让开发者更加明确容器保存的是什么类型的数据。

下面是一段关于泛型的代码：

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class hzlTest{
    public static void main(String[] args){
        List list1 = new ArrayList();              //无泛型创建链表对象
        list1.add("a");
        list.add("b");
        Iterator it = list1.iterator();            //获取链表迭代器
        while(it.hasNext()){
            Object obj = it.next();                //获取元素，类型为 Object
            String val = (String) obj;             //类型转换
```

```
            System.out.println(val);
        }
        List<String> list2 = new ArrayList<String>();     //String 泛型链表对象
        list2.add("a");
        list2.add("b");
        for(String str : list2){                //用 foreach 进行遍历，类型直接转换为 String
            System.out.println(str);
        }
    }
}
```

23. 两个对象值相同(x.equals(y) == true)，但却有不同的 hashCode，这句话对吗？

对。如果对象要保存在 HashSet 或 HashMap 中，它们的 equals 相等，那么，它们的 hashCode 值就必须相等。如果不是要保存在 HashSet 或 HashMap， 则与 hashCode 没有什么关系了。这时候 hashcode 不等是可以的，例如 arrayList 存储的对象就不用实现 hashCode，当然，我们没有理由不实现，通常都会去实现的。

24. 说出一些常用的类，包，接口，请各举 5 个

要让人家感觉你对 java ee 开发很熟，所以，不能仅仅只列 core java 中的那些东西，要多 列你在做 ssh 项目中涉及的那些东西。就写你最近写的那些程序中涉及的那些类。

常用的类：BufferedReader，BufferedWriter， FileReader， FileWirter， String， Integer，java.util.Date，System，Class，List HashMap

常用的包： java.lang， java.io， java.util， java.sql， javax.servlet， org.hibernate ，org.apache.strtuts.action

常用的接口： Remote， List，Map，Document，NodeList，Servlet，HttpServletRequest，HttpServletResponse，Transaction(Hibernate)，Session(Hibernate)，HttpSession

25. 反射的概念,哪儿需要反射机制，反射的性能，如何优化？

在运行状态中，对于任意的一个类，都能够知道这个类的所有属性和方法，对任意一个对象，都能够通过反射机制调用一个类的任意方法，这种动态获取类信息及动态调用类对象方法的功能称为 java 的反射机制。

反射的作用：

① 动态地创建类的实例，将类绑定到现有的对象中，或从现有的对象中获取类型。

② 应用程序需要在运行时从某个特定的程序集中载入一个特定的类。

五、垃圾收集器(Garbage Collectors)

1. Java 中垃圾回收有什么目的？什么时候进行垃圾回收？

垃圾回收的目的是识别并且丢弃应用不再使用的对象来释放和重用资源。

GC 是垃圾收集的意思（Gabage Collection）,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。

2. System.gc()和 Runtime.gc()会做什么事情？

这两个方法用来提示 JVM 要进行垃圾回收。但是，立即开始还是延迟进行垃圾回收是取决于 JVM 的。

3. finalize()方法什么时候被调用？析构函数(finalization)的目的是什么？

在释放对象占用的内存之前，垃圾收集器会调用对象的 finalize()方法。一般建议在该方法中释放对象持有的资源。

4. 如果对象的引用被置为 null，垃圾收集器是否会立即释放对象占用的内存？

不会，在下一个垃圾回收周期中，这个对象将是可被回收的。

5. Java 堆的结构是什么样子的？什么是堆中的永久代(Perm Gen space)？

JVM 的堆是运行时数据区，所有类的实例和数组都是在堆上分配内存。它在 JVM 启动的时候被创建。对象所占的堆内存是由自动内存管理系统也就是垃圾收集器回收。

堆内存是由存活和死亡的对象组成的。存活的对象是应用可以访问的，不会被垃圾回收。死亡的对象是应用不可访问尚且还没有被垃圾收集器回收掉的对象。一直到垃圾收集器把这些对象回收掉之前，他们会一直占据堆内存空间。

6. 串行(serial)收集器和吞吐量(throughput)收集器的区别是什么？

吞吐量收集器使用并行版本的新生代垃圾收集器，它用于中等规模和大规模数据的应用程序。而串行收集器对大多数的小应用(在现代处理器上需要大概 100M 左右的内存)就足够了。

7. 在 Java 中，对象什么时候可以被垃圾回收？

当对象对当前使用这个对象的应用程序变得不可触及的时候，这个对象就可以被回收了。

8. JVM 的永久代中会发生垃圾回收么？

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免 Full GC 是非常重要的原因。请参考下 Java8：从永久代到元数据区

(备注：Java8 中已经移除了永久代，新加了一个叫做元数据区的 native 内存区)

9. Java 程序为什么无须 delete 语句进行内存回收？

Java 的堆内存数据的释放功能是由垃圾回收器自动进行的，无须程序员显示的调用 delete 方法。该机制有效地避免了因为程序员忘记释放内存造成的内存溢出的错误，相对于 C++等需要显示释放内存的语言，是一种巨大的改进。

10. 垃圾回收的优点和原理。并考虑 2 种回收机制。

Java 中一个显著的特点就是引入了垃圾回收机制，使 c++程序员最头疼的内存管理问题迎刃而解，它使得 Java 程序员在编写程序的时候不再需要考虑内存管理。由于有个垃圾回收机制，Java 中的对象不再有"作用域"的概念，只有对象的引用才有"作用域"。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级别的线程运行，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。

回收机制有分带回收、复制垃圾回收、标记-清除垃圾回收、标记-压缩垃圾回收、增量垃圾回收。

11. 垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？

对于 GC 来说，当程序员创建对象时，GC 就开始监控这个对象的地址、大小以及使用情况。通常，GC 采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当 GC 确定一些对象为"不可达"时，GC 就有责任回收这些内存空间。

可以。

程序员可以手动执行 System.gc()，通知 GC 运行，但是 Java 语言规范并不保证 GC 一定会执行。

六、异常处理

1. Java 中的两种异常类型是什么？他们有什么区别？

Java 中有两种异常：受检查的(checked)异常和不受检查的(unchecked)异常。

不受检查的异常不需要在方法或者是构造函数上声明，就算方法或者是构造函数的执行可能会抛出这样的异常，并且不受检查的异常可以传播到方法或者是构造函数的外面。

相反，受检查的异常必须要用 throws 语句在方法或者是构造函数上声明。这里有 Java 异常处理的一些小建议。

2. Java 中 Exception 和 Error 有什么区别？

Exception 和 Error 都是 Throwable 的子类。Exception 用于用户程序可以捕获的异常情况。Error 定义了不期望被用户程序捕获的异常。

注明：

① Throwable 类是 Java 语言中所有错误或异常的超类。 它的两个子类是 Error 和 Exception；

② Error 是 Throwable 的子类，用于指示合理的应用程序不应该试图捕获的严重问题。大多数这样的错误都是异常条件。虽然 ThreadDeath 错误是一个“正规”的条件，但它也是 Error 的子类，因为大多数应用程序都不应该试图捕获它。在执行该方法期间，无需在其 throws 子句中声明可能抛出但是未能捕获的 Error 的任何子类，因为这些错误可能是再也不会发生的异常条件。

③ Exception 类及其子类是 Throwable 的一种形式，它指出了合理的应用程序想要捕获的条件。

④ RuntimeException 是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类。可能在执行方法期间抛出但未被捕获的， RuntimeException 的任何子类都无需在 throws 子句中进行声明，它是 Exception 的子类。常见的 RuntimeException 有 NullPointerException（空指针异常）、ArrayIndexOutOfBoundsException（数组脚本越界）、ClassCastException（类转换异常）。

3. throw 和 throws 有什么区别？

throw 关键字用来在程序中明确的抛出异常，相反，throws 语句用来表明方法不能处理的异常。每一个方法都必须要指定哪些异常不能处理，所以方法的调用者才能够确保处理可能发生的异常，多个异常是用逗号分隔的。

4. 异常处理的时候，finally 代码块的重要性是什么？

无论是否抛出异常，finally 代码块总是会被执行。就算是没有 catch 语句同时又抛出异常的情况下，finally 代码块仍然会被执行。最后要说的是，finally 代码块主要用来释放资源，比如：I/O 缓冲区，数据库连接。

5. 异常处理完成以后，Exception 对象会发生什么变化？

Exception 对象会在下一个垃圾回收过程中被回收掉.

6. finally 代码块和 finalize()方法有什么区别？

无论是否抛出异常，finally 代码块都会执行，它主要是用来释放应用占用的资源。finalize()方法是 Object 类的一个 protected 方法，它是在对象被垃圾回收之前由 Java 虚拟机来调用的。

7. final，finally，finalize 的区别。

final 用于声明属性，方法和类，分别表示属性不可变，方法不可覆盖，类不可继承。内部类要访问局部变量，局部变量必须定义成 final 类型。

finally 用在 try/catch 语句块中， 是异常处理语句结构的一部分，表示总会执行。

finalize 是 Object 类的一个方法，当垃圾收集器确定再无任何引用指向某个对象实例时，就会在销毁该对象之前调用 finalize()方法，一般用于清理资源，如关闭文件等。JVM 不保证此方法总被调用。

8. JAVA 语言如何进行异常处理，关键字：throws，throw，try，catch，finally 分别代表什么意义？在 try 块中可以抛出异常吗？

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，并提供了良好的接口。在 Java 中，每个异常都是一个对象，它是 Throwable 类或其它子类的实例。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进行处理。Java 的异常处理是通过 5 个关键词来实现的：

try、catch、throw、throws 和 finally。一般情况下是用 try 来执行一段程序，如果出现异常，系统会抛出（throws）一个异常，这时候你可以通过它的类型来捕捉（catch）它，或最后（finally）由缺省处理器来处理。用 try 来指定一块预防所有"异常"的程序。紧跟在 try 程序后面，应包含一个 catch 子句来指定你想要捕捉的"异常"的类型。throw 语句用来明确地抛出一个"异常"。throws 用来标明一个成员函数可能抛出的各种"异常"。Finally 为确保一段代码不管发生什么"异常"都被执行一段代码。可以在一个成员函数调用的外面写一个 try 语句，在这个成员函数内部写另一个 try 语句保护其他代码。每当遇到一个 try 语句，"异

常"的框架就放到堆栈上面，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，堆栈就会展开，直到遇到有处理这种"异常"的 try 语句。

七、Java 小应用程序(Applet)

1. 什么是 Applet？

Java applet 是能够被包含在 HTML 页面中并且能被启用了 java 的客户端浏览器执行的程序。Applet 主要用来创建动态交互的 web 应用程序。

2. 解释一下 Applet 的生命周期

applet 可以经历下面的状态：

Init：每次被载入的时候都会被初始化。

Start：开始执行 applet。

Stop：结束执行 applet。

Destroy：卸载 applet 之前，做最后的清理工作。

3. 当 applet 被载入的时候会发生什么？

首先，创建 applet 控制类的实例，然后初始化 applet，最后开始运行。

4. Applet 和普通的 Java 应用程序有什么区别？

applet 是运行在启用了 java 的浏览器中，Java 应用程序是可以在浏览器之外运行的独立的 Java 程序。但是，它们都需要有 Java 虚拟机。

进一步来说，Java 应用程序需要一个有特定方法签名的 main 函数来开始执行。Java applet 不需要这样的函数来开始执行。最后，Java applet 一般会使用很严格的安全策略，Java 应用一般使用比较宽松的安全策略。

5. Java applet 有哪些限制条件？

主要是由于安全的原因，给 applet 施加了以下的限制：

- applet 不能够载入类库或者定义本地方法。
- applet 不能在宿主机上读写文件。
- applet 不能读取特定的系统属性。
- applet 不能发起网络连接，除非是跟宿主机。
- applet 不能够开启宿主机上其他任何的程序。

6. 什么是不受信任的 applet？

不受信任的 applet 是不能访问或是执行本地系统文件的 Java applet，默认情况下，所有下载的 applet 都是不受信任的。

7. 从网络上加载的 applet 和从本地文件系统加载的 applet 有什么区别？

当 applet 是从网络上加载的时候，applet 是由 applet 类加载器载入的，它受 applet 安全管理器的限制。

当 applet 是从客户端的本地磁盘载入的时候，applet 是由文件系统加载器载入的。

从文件系统载入的 applet 允许在客户端读文件，写文件，加载类库，并且也允许执行其他程序，但是，却通不过字节码校验。

8. applet 类加载器是什么？它会做哪些工作？

当 applet 是从网络上加载的时候，它是由 applet 类加载器载入的。类加载器有自己的 java 名称空间等级结构。类加载器会保证来自文件系统的类有唯一的名称空间，来自网络资源的类有唯一的名称空间。

当浏览器通过网络载入 applet 的时候，applet 的类被放置于和 applet 的源相关联的私有的名称空间中。然后，那些被类加载器载入进来的类都是通过了验证器验证的。验证器会检查类文件格式是否遵守 Java 语言规范，确保不会出现堆栈溢出(stack overflow)或者下溢(underflow)，传递给字节码指令的参数是正确的。

9. applet 安全管理器是什么？它会做哪些工作？

applet 安全管理器是给 applet 施加限制条件的一种机制。浏览器可以只有一个安全管理器。安全管理器在启动的时候被创建，之后不能被替换覆盖或者是扩展。

八、Swing

1. 弹出式选择菜单(Choice)和列表(List)有什么区别

Choice 是以一种紧凑的形式展示的，需要下拉才能看到所有的选项。Choice 中一次只能选中一个选项。List 同时可以有多个元素可见，支持选中一个或者多个元素。

2. 什么是布局管理器？

布局管理器用来在容器中组织组件。

3. 滚动条(Scrollbar)和滚动面板(JScrollPane)有什么区别？

Scrollbar 是一个组件，不是容器。而 ScrollPane 是容器。ScrollPane 自己处理滚动事件。

4. 哪些 Swing 的方法是线程安全的？

只有 3 个线程安全的方法： repaint(), revalidate(), and invalidate()。

5. 说出三种支持重绘(painting)的组件。

Canvas, Frame, Panel 和 Applet 支持重绘。

6. 什么是裁剪(clipping)？

限制在一个给定的区域或者形状的绘图操作就做裁剪。

7. MenuItem 和 CheckboxMenuItem 的区别是什么？

CheckboxMenuItem 类继承自 MenuItem 类，支持菜单选项可以选中或者不选中。

8. 边缘布局(BorderLayout)里面的元素是如何布局的？

BorderLayout 里面的元素是按照容器的东西南北中进行布局的。

9. 网格包布局(GridBagLayout)里面的元素是如何布局的？

GridBagLayout 里面的元素是按照网格进行布局的。不同大小的元素可能会占据网格的多于 1 行或一列。因此，行数和列数可以有不同的大小。

10. Window 和 Frame 有什么区别？

Frame 类继承了 Window 类，它定义了一个可以有菜单栏的主应用窗口。

11. 裁剪(clipping)和重绘(repainting)有什么联系？

当窗口被 AWT 线程进行重绘的时候，它会把裁剪区域设置成需要重绘的窗口的区域。

12. 事件监听器接口(event-listener interface)和事件适配器(event-adapter)有什么关系？

事件监听器接口定义了对特定的事件，事件处理器必须要实现的方法。事件适配器给事件监听器接口提供了默认的实现。

13. GUI 组件如何来处理它自己的事件？

GUI 组件可以处理它自己的事件，只要它实现相对应的事件监听器接口，并且把自己作为事件监听器。

14. Java 的布局管理器比传统的窗口系统有哪些优势？

Java 使用布局管理器以一种一致的方式在所有的窗口平台上摆放组件。因为布局管理器不会和组件的绝对大小和位置相绑定，所以他们能够适应跨窗口系统的特定平台的不同。

15. Java 的 Swing 组件使用了哪种设计模式？

Java 中的 Swing 组件使用了 MVC(视图-模型-控制器)设计模式。

九、JDBC

1. 什么是 JDBC?

JDBC 是允许用户在不同数据库之间做选择的一个抽象层。JDBC 允许开发者用 JAVA 写数据库应用程序，而不需要关心底层特定数据库的细节。

2. 解释下驱动(Driver)在 JDBC 中的角色。

JDBC 驱动提供了特定厂商对 JDBC API 接口类的实现，驱动必须要提供 java.sql 包下面这些类的实现：Connection, Statement, PreparedStatement, CallableStatement, ResultSet 和 Driver。

3. Class.forName()方法有什么作用?

这个方法用来载入跟数据库建立连接的驱动。

4. Statement 与 PreparedStatement 的区别,什么是 SQL 注入，如何防止 SQL 注入?

① PreparedStatement 支持动态设置参数，Statement 不支持。

②PreparedStatement 可避免如类似单引号的编码麻烦，Statement 不可以。

③ PreparedStatement 支持预编译，Statement 不支持。

④ 在 sql 语句出错时 PreparedStatement 不易检查，而 Statement 则更便于查错。

⑤ PreparedStatement 可防止 Sql 注入，更加安全，而 Statement 不行。

Sql 注入是指 通过 sql 语句的拼接达到无参数查询数据库数据目的的方法。

如将要执行的 sql 语句为 select * from table where name = "+appName+"，利用 appName 参数值的输入来生成恶意的 sql 语句，如将['or'1'='1'] 传入可在数据库中执行。

因此可以采用 PrepareStatement 来避免 Sql 注入，在服务器端接收参数数据后，进行验证，此时 PrepareStatement 会自动检测，而 Statement 不行，需要手工检测。

5.什么时候使用 CallableStatement? 用来准备 CallableStatement 的方法是什么?

CallableStatement 用来执行存储过程。存储过程是由数据库存储和提供的。存储过程可以接受输入参数，也可以有返回结果。非常鼓励使用存储过程，因为它提供了安全性和模块化。准备一个 CallableStatement 的方法是：CallableStament.prepareCall();

6. 数据库连接池是什么意思？

像打开关闭数据库连接这种和数据库的交互可能是很费时的，尤其是当客户端数量增加的时候，会消耗大量的资源，成本是非常高的。可以在应用服务器启动的时候建立很多个数据库连接并维护在一个池中。连接请求由池中的连接提供。在连接使用完毕以后，把连接归还到池中，以用于满足将来更多的请求。

数据库连接建立一个“缓冲池”。预先在缓冲池中放入一定数量的连接，当需要建立数据库连接时，只需从“缓冲池”中取出一个，使用完毕之后再放回去。我们可以通过设定连接池最大连接数来防止系统无尽的与数据库连接。更为重要的是我们可以通过连接池的管理机制监视数据库的连接的数量、使用情况，为系统开发、测试及性能调整提供依据。

7. 数据库的事务属性？

① 原子性(Atomicity）

所谓原子性就是将一组操作作为一个操作单元，是原子操作，即要么全部执行，要么全部不执行。

② 一致性（Consistency）

事务的一致性指的是在一个事务执行之前和执行之后数据库都必须处于一致性状态。如果事务成功地完成，那么系统中所有变化将正确地应用，系统处于有效状态。如果在事务中出现错误，那么系统中的所有变化将自动地回滚，系统返回到原始状态。

③ 隔离性（Isolation）

隔离性指并发的事务是相互隔离的。即一个事务内部的操作及正在操作的数据必须封锁起来，不被其它企图进行修改的事务看到。

④ 持久性（Durability）

持久性是指一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来的其他操作和数据库故障不应该对其有任何影响。即一个事务一旦提交，DBMS（Database Management System）保证它对数据库中数据的改变应该是永久性的，持久性通过数据库备份和恢复来保证。

8. 如何查 200 到 300 行的记录，可以通过 top 关键字辅助

select top 100 * from table where id is not in (select top 200 id from table);

查询 n 到 m 行记录的通用公式：select top m * from table where id is not in (select top n * from table)

9. 事务的隔离级别？

数据库事务的隔离级别有 4 个，由低到高依次为 Read uncommitted、Read committed、Repeatable read、Serializable，这四个级别可以逐个解决脏读、不可重复读、幻读这几类问题。

√：可能出现　　×：不会出现

	脏读	不可重复读	幻读
Read uncommitted	√	√	√
Read committed	×	√	√
Repeatable read	×	×	√
Serializable	×	×	×

Read uncommitted 读未提交

公司发工资了，领导把 5000 元打到 singo 的账号上，但是该事务并未提交，而 singo 正好去查看账户，发现工资已经到账，是 5000 元整，非常高兴。可是不幸的是，领导发现发给 singo 的工资金额不对，是 2000 元，于是迅速 回滚了事务，修改金额后，将事务提交，最后 singo 实际的工资只有 2000 元，singo 空欢喜一场。

出现上述情况，即我们所说的脏读，两个并发的事务，“事务 A：领导给 singo 发工资”、“事务 B：singo 查询工资账户”，事务 B 读取了事务 A 尚未提交的数据。

当隔离级别设置为 Read uncommitted 时，就可能出现脏读，如何避免脏读，请看下一个隔离级别。

Read committed 读提交

singo 拿着工资卡去消费，系统读取到卡里确实有 2000 元，而此时她的老婆也正好在网上转账，把 singo 工资卡的 2000 元转到另一账户，并在 singo 之前提交了事务，当 singo 扣款时，系统检查到 singo 的工资卡已经没有 钱，扣款失败，singo 十分纳闷，明明卡里有钱，为何.....

出现上述情况，即我们所说的不可重复读，两个并发的事务，“事务 A：singo 消费”、“事务 B：singo 的老婆网上转账”，事务 A 事先读取了数据，事务 B 紧接了更新了数据，并提交了事务，而事务 A 再次读取该数据时，数据已经发生了改变。

当隔离级别设置为 Read committed 时，避免了脏读，但是可能会造成不可重复读。

大多数数据库**的默认级别**就是 Read committed，比如 Sql Server , Oracle。如何解决不可重复读这一问题，请看下一个隔离级别。

Repeatable read 重复读

当隔离级别设置为 Repeatable read 时，可以避免不可重复读。当 singo 拿着工资卡去消费时，一旦系统开始读取工资卡信息（即事务开始），singo 的老婆就不可能对该记录进行修改，也就是 singo 的老婆不能在此时转账。

虽然 Repeatable read 避免了不可重复读，但还有可能出现幻读。

singo 的老婆工作在银行部门，她时常通过银行内部系统查看 singo 的信用卡消费 记录。有一天，她正在查询到 singo 当月信用卡的总消费金额（select sum(amount) from transaction where month = 本月）为 80 元，而 singo 此时正好在外面胡吃海塞后在收银台买单，消费 1000 元，即新增了一条 1000 元的消费记录（insert transaction ... ），并提交了事务，随后 singo 的老婆将 singo 当月信用卡消费的明细打印到 A4 纸上，却发现消费总额为 1080 元，singo 的老婆很诧异，以为出 现了幻觉，幻读就这样产生了。

注：Mysql 的默认隔离级别就是 Repeatable read。

Serializable 序列化

Serializable 是最高的事务隔离级别，同时代价也花费最高，性能很低，一般很少使用，在该级别下，事务顺序执行，不仅可以避免脏读、不可重复读，还避免了幻像读。

十、远程方法调用(RMI)

1. 什么是 RMI？

Java 远程方法调用(Java RMI)是 Java API 对远程过程调用(RPC)提供的面向对象的等价形式，支持直接传输序列化的 Java 对象和分布式垃圾回收。远程方法调用可以看做是激活远程正在运行的对象上的方法的步骤。RMI 对调用者是位置透明的，因为调用者感觉方法是执行在本地运行的对象上的。看下 RMI 的一些注意事项。

2. RMI 体系结构的基本原则是什么？

RMI 体系结构是基于一个非常重要的行为定义和行为实现相分离的原则。RMI 允许定义行为的代码和实现行为的代码相分离，并且运行在不同的 JVM 上。

3. RMI 体系结构分哪几层？

RMI 体系结构分以下几层：

存根和骨架层(Stub and Skeleton layer)：这一层对程序员是透明的，它主要负责拦截客户端发出的方法调用请求，然后把请求重定向给远程的 RMI 服务。

远程引用层(Remote Reference Layer)：RMI 体系结构的第二层用来解析客户端对服务端远程对象的引用。这一层解析并管理客户端对服务端远程对象的引用。连接是点到点的。

传输层(Transport layer)：这一层负责连接参与服务的两个 JVM。这一层是建立在网络上机器间的 TCP/IP 连接之上的。它提供了基本的连接服务，还有一些防火墙穿透策略。

4. RMI 中的远程接口(Remote Interface)扮演了什么样的角色？

远程接口用来标识哪些方法是可以被非本地虚拟机调用的接口。远程对象必须要直接或者是间接实现远程接口。实现了远程接口的类应该声明被实现的远程接口，给每一个远程对象定义构造函数，给所有远程接口的方法提供实现。

5. java.rmi.Naming 类扮演了什么样的角色？

java.rmi.Naming 类用来存储和获取在远程对象注册表里面的远程对象的引用。Naming 类的每一个方法接收一个 URL 格式的 String 对象作为它的参数。

6. RMI 的绑定(Binding)是什么意思？

绑定是为了查询找远程对象而给远程对象关联或者是注册以后会用到的名称的过程。远程对象可以使用 Naming 类的 bind()或者 rebind()方法跟名称相关联。

7. Naming 类的 bind()和 rebind()方法有什么区别？

bind()方法负责把指定名称绑定给远程对象，rebind()方法负责把指定名称重新绑定到一个新的远程对象。如果那个名称已经绑定过了，先前的绑定会被替换掉。

8. 让 RMI 程序能正确运行有哪些步骤？

为了让 RMI 程序能正确运行必须要包含以下几个步骤：

编译所有的源文件。

使用 rmic 生成 stub。

启动 rmiregistry。

启动 RMI 服务器。

运行客户端程序。

9. RMI 的 stub 扮演了什么样的角色？

远程对象的 stub 扮演了远程对象的代表或者代理的角色。调用者在本地 stub 上调用方法，它负责在远程对象上执行方法。当 stub 的方法被调用的时候，会经历以下几个步骤：

初始化到包含了远程对象的 JVM 的连接。

序列化参数到远程的 JVM。

等待方法调用和执行的结果。

反序列化返回的值或者是方法没有执行成功情况下的异常。

把值返回给调用者。

10. 什么是分布式垃圾回收(DGC)？它是如何工作的？

DGC 叫做分布式垃圾回收。RMI 使用 DGC 来做自动垃圾回收。因为 RMI 包含了跨虚拟机的远程对象的引用，垃圾回收是很困难的。DGC 使用引用计数算法来给远程对象提供自动内存管理。

11. RMI 中使用 RMI 安全管理器(RMISecurityManager)的目的是什么？

RMISecurityManager 使用下载好的代码提供可被 RMI 应用程序使用的安全管理器。如果没有设置安全管理器，RMI 的类加载器就不会从远程下载任何的类。

12. 解释下 Marshalling 和 demarshalling。

当应用程序希望把内存对象跨网络传递到另一台主机或者是持久化到存储的时候，就必须要把对象在内存里面的表示转化成合适的格式。这个过程就叫做 Marshalling，反之就是 demarshalling。

13. 解释下 Serialization 和 Deserialization。

Java 提供了一种叫做对象序列化的机制，他把对象表示成一连串的字节，里面包含了对象的数据，对象的类型信息，对象内部的数据的类型信息等等。因此，序列化可以看成是为了把对象存储在磁盘上或者是从磁盘上读出来并重建对象而把对象扁平化的一种方式。反序列化是把对象从扁平状态转化成活动对象的相反的步骤。

十一、Servlet

1.什么是 Servlet？

Servlet 是用来处理客户端请求并产生动态网页内容的 Java 类。Servlet 主要是用来处理或者是存储 HTML 表单提交的数据，产生动态内容，在无状态的 HTTP 协议下管理状态信息。

2. 说一下 Servlet 的体系结构。

所有的 Servlet 都必须要实现的核心的接口是 javax.servlet.Servlet。每一个 Servlet 都必须要直接或者是间接实现这个接口，或者是继承 javax.servlet.GenericServlet 或者 javax.servlet.http.HTTPServlet。最后，Servlet 使用多线程可以并行的为多个请求服务。

3. Applet 和 Servlet 有什么区别？

Applet 是运行在客户端主机的浏览器上的客户端 Java 程序。而 Servlet 是运行在 web 服务器上的服务端的组件。applet 可以使用用户界面类，而 Servlet 没有用户界面，相反，Servlet 是等待客户端的 HTTP 请求，然后为请求产生响应。

4.GenericServlet 和 HttpServlet 有什么区别？

GenericServlet 是一个通用的协议无关的 Servlet，它实现了 Servlet 和 ServletConfig 接口。继承自 GenericServlet 的 Servlet 应该要覆盖 service()方法。最后，为了开发一个能用在网页上服务于使用 HTTP 协议请求的 Servlet，你的 Servlet 必须要继承自 HttpServlet。

5.解释下 Servlet 的生命周期。

大致分为 4 步： Servlet 类加载-->实例化-->服务-->销毁

对每一个客户端的请求，Servlet 引擎载入 Servlet，调用它的 init()方法，完成 Servlet 的初始化。然后，Servlet 对象通过为每一个请求单独调用 service()方法来处理所有随后来自客户端的请求，最后，调用 Servlet(这里应该是 Servlet 而不是 server())的 destroy()方法把 Servlet 删除掉。

6. doGet()方法和 doPost()方法有什么区别？

doGet：GET 方法会把名值对追加在请求的 URL 后面。因为 URL 对字符数目有限制，进而限制了用在客户端请求的参数值的数目。并且请求中的参数值是可见的，因此，敏感信息不能用这种方式传递。

doPost：POST 方法通过把请求参数值放在请求体中来克服 GET 方法的限制，因此，可以发送的参数的数目是没有限制的。最后，通过 POST 请求传递的敏感信息对外部客户端是不可见的。

post 和 get 提交的区别？

① Get 是从服务器端获取数据，Post 则是向服务器端发送数据。

② 在客户端，Get 方式通过 URL 提交数据，在 URL 地址栏可以看到请求消息，该消息被编码过；Post 数据则是放在 Html header 内提交。

③ 对于 Get 方式，服务器端用 Request.QueryString 获取变量的值；对用 Post 方式，服务器端用 Request.Form 获取提交的数据值。

④ Get 方式提交的数据最多 1024 字节，而 Post 则没有限制。

⑤ Get 方式提交的参数及参数值会在地址栏显示，不安全，而 Post 不会，比较安全。

7. 什么是 Web 应用程序？

Web 应用程序是对 Web 或者是应用服务器的动态扩展。有两种类型的 Web 应用：面向表现的和面向服务的。面向表现的 Web 应用程序会产生包含了很多种标记语言和动态内容的交互的 web 页面作为对请求的响应。而面向服务的 Web 应用实现了 Web 服务的端点(endpoint)。一般来说，一个 Web 应用可以看成是一组安装在服务器 URL 名称空间的特定子集下面的 Servlet 的集合。

8. 什么是服务端包含(Server Side Include)？

服务端包含(SSI)是一种简单的解释型服务端脚本语言，大多数时候仅用在 Web 上，用 servlet 标签嵌入进来。SSI 最常用的场景把一个或多个文件包含到 Web 服务器的一个 Web 页面中。当浏览器访问 Web 页面的时候，Web 服务器会用对应的 servlet 产生的文本来替换 Web 页面中的 servlet 标签。

9. 什么是 Servlet 链(Servlet Chaining)？

Servlet 链是把一个 Servlet 的输出发送给另一个 Servlet 的方法。第二个 Servlet 的输出可以发送给第三个 Servlet，依次类推。链条上最后一个 Servlet 负责把响应发送给客户端。

10. 如何知道是哪一个客户端的机器正在请求你的 Servlet？

ServletRequest 类可以找出客户端机器的 IP 地址或者是主机名。getRemoteAddr()方法获取客户端主机的 IP 地址，getRemoteHost()可以获取主机名。

11. HTTP 响应的结构是怎么样的？

HTTP 响应由三个部分组成：

状态码(Status Code)：描述了响应的状态。可以用来检查是否成功的完成了请求。请求失败的情况下，状态码可用来找出失败的原因。如果 Servlet 没有返回状态码，默认会返回成功的状态码 HttpServletResponse.SC_OK。

HTTP 头部(HTTP Header)：它们包含了更多关于响应的信息。比如：头部可以指定认为响应过期的过期日期，或者是指定用来给用户安全的传输实体内容的编码格式。如何在 Serlet 中检索 HTTP 的头部看这里。

主体(Body)：它包含了响应的内容。它可以包含 HTML 代码，图片等等。主体是由传输在 HTTP 消息中紧跟在头部后面的数据字节组成的。

［**注**］HTTP 报文包含内容

①request line

②header line 用来说明服务器要使用的附加信息

③blank line 这是 Http 的规定，必须空一行

④request body 请求的内容数据

12. 什么是 cookie？session 和 cookie 有什么区别？

cookie 是 Web 服务器发送给浏览器的一块信息。浏览器会在本地文件中给每一个 Web 服务器存储 cookie。以后浏览器在给特定的 Web 服务器发请求的时候，同时会发送所有为该服务器存储的 cookie。下面列出了 session 和 cookie 的区别：

无论客户端浏览器做怎么样的设置，session 都应该能正常工作。客户端可以选择禁用 cookie，但是，session 仍然是能够工作的，因为客户端无法禁用服务端的 session。

在存储的数据量方面 session 和 cookies 也是不一样的。session 能够存储任意的 Java 对象，cookie 只能存储 String 类型的对象。

①Cookie 是客户端的存储空间，由浏览器维护；Session 由应用服务器维护的一个服务器端的存储空间。

②Cookie 中保存的是字符串；Session 中保存的是对象。

③ cookie 数据存放在客户的浏览器上，session 数据放在服务器上。

④ cookie 不是很安全，别人可以分析存放在本地的 cookie 并进行 cookie 欺骗考虑到安全应当使用 session。

⑤ session 会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能，考虑到减轻服务器性能方面，应当使用 cookie。

⑥ Session 和 Cookie 不能跨窗口使用，每打开一个浏览器系统会赋予一个 SessionID，此时的 SessionID 不同，若要完成跨浏览器访问数据，可以使用 Application。

⑦ Session、Cookie 都有失效时间，过期后会自动删除，减少系统开销。

13. 浏览器和 Servlet 通信使用的是什么协议？

浏览器和 Servlet 通信使用的是 HTTP 协议。

14. 什么是 HTTP 隧道？

HTTP 隧道是一种利用 HTTP 或者是 HTTPS 把多种网络协议封装起来进行通信的技术。因此，HTTP 协议扮演了一个打通用于通信的网络协议的管道的包装器的角色。把其他协议的请求掩盖成 HTTP 的请求就是 HTTP 隧道。

15. sendRedirect()和 forward()方法有什么区别？

sendRedirect()方法会创建一个新的请求，而 forward()方法只是把请求转发到一个新的目标上。重定向(redirect)以后，之前请求作用域范围以内的对象就失效了，因为会产生一个新的请求，而转发(forwarding)以后，之前请求作用域范围以内的对象还是能访问的。一般认为 sendRedirect()比 forward()要慢。

原理：forward 是服务器请求资源，服务器直接访问目标地址的 URL，把那个 URL 的响应内容读取过来，然后再将这些内容返回给浏览器，浏览器根本不知道服务器发送的这些内容是从哪来的，所以地址栏还是原来的地址。

redirect 是服务器根据逻辑，发送一个状态码，告诉浏览器重新去请求的那个地址，浏览器会用刚才请求的所有参数重新发送新的请求，所以 session，request 参数都可以获取

区别：①前者仅是容器中控制权的转向，在客户端浏览器地址栏中不会显示出转向后的地址。

② 后者则是完全的跳转，浏览器将会得到跳转的地址，并重新发送请求链接。这样，从浏览器的地址栏中可以看到跳转后的链接地址。

③ 前者更加高效，在前者可以满足需要时，尽量使用 forward()方法，并且，这样也有助于隐藏实际的链接。

④ 在有些情况下，比如需要跳转到一个其它服务器上的资源，则必须使 sendRedirect()方法。

16. 什么是 URL 编码和 URL 解码？

URL 编码是负责把 URL 里面的空格和其他的特殊字符替换成对应的十六进制表示，反之就是解码。

17. Tomcat，Apache，JBoss 的区别？

① Apache 是 Http 服务器，Tomcat 是 web 服务器，JBoss 是应用服务器。

② Apache 解析静态的 html 文件；Tomcat 可解析 jsp 动态页面、也可充当 servlet 容器。

18. HTTPS 和 HTTP 的区别？

http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 **80**，后者是 **443**。

http 的连接很简单，是无状态的。HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比 http 协议安全，被广泛用于万维网上安全敏感的通讯，例如交易支付方面。

十二、JSP

1. 什么是 JSP 页面？

JSP 页面是一种包含了静态数据和 JSP 元素两种类型的文本的文本文档。静态数据可以用任何基于文本的格式来表示，比如：HTML 或者 XML。JSP 是一种混合了静态内容和动态产生的内容的技术。

2. JSP 请求是如何被处理的？

浏览器首先要请求一个以.jsp 扩展名结尾的页面，发起 JSP 请求，然后，Web 服务器读取这个请求，使用 JSP 编译器把 JSP 页面转化成一个 Servlet 类。需要注意的是，只有当第一次请求页面或者是 JSP 文件发生改变的时候 JSP 文件才会被编译，然后服务器调用 servlet 类，处理浏览器的请求。一旦请求执行结束，servlet 会把响应发送给客户端。这里看下如何在 JSP 中获取请求参数。

3. JSP 有什么优点？

下面列出了使用 JSP 的优点：

①JSP 页面是被动态编译成 Servlet 的，因此，开发者可以很容易的更新展现代码。

②JSP 页面可以被预编译。

③JSP 页面可以很容易的和静态模板结合，包括：HTML 或者 XML，也可以很容易的和产生动态内容的代码结合起来。

④开发者可以提供让页面设计者以类 XML 格式来访问的自定义的 JSP 标签库。

⑤开发者可以在组件层做逻辑上的改变，而不需要编辑单独使用了应用层逻辑的页面。

4. 什么是 JSP 指令(Directive)？JSP 中有哪些不同类型的指令？

Directive 是当 JSP 页面被编译成 Servlet 的时候，JSP 引擎要处理的指令。Directive 用来设置页面级别的指令，从外部文件插入数据，指定自定义的标签库。Directive 是定义在<%@ 和 %>之间的。下面列出了不同类型的 Directive：

包含指令(Include directive)：用来包含文件和合并文件内容到当前的页面。

页面指令(Page directive)：用来定义 JSP 页面中特定的属性，比如错误页面和缓冲区。

Taglib 指令： 用来声明页面中使用的自定义的标签库。

5. 什么是 JSP 动作(JSP action)？

JSP 动作以 XML 语法的结构来控制 Servlet 引擎的行为。当 JSP 页面被请求的时候，JSP 动作会被执行。它们可以被动态的插入到文件中，重用 JavaBean 组件，转发用户到其他的页面，或者是给 Java 插件产生 HTML 代码。下面列出了可用的动作：

jsp:include-当 JSP 页面被请求的时候包含一个文件。

jsp:useBean-找出或者是初始化 Javabean。

jsp:setProperty-设置 JavaBean 的属性。

jsp:getProperty-获取 JavaBean 的属性。

jsp:forward-把请求转发到新的页面。

jsp:plugin-产生特定浏览器的代码。

6.什么是 Scriptlets？

JSP 技术中，scriptlet 是嵌入在 JSP 页面中的一段 Java 代码。scriptlet 是位于标签内部的所有的东西，在标签与标签之间，用户可以添加任意有效的 scriplet。

7. 声明(Decalaration)在哪里？

声明跟 Java 中的变量声明很相似，它用来声明随后要被表达式或者 scriptlet 使用的变量。添加的声明必须要用开始和结束标签包起来。

8.什么是表达式(Expression)？

【列表很长，可以分上、中、下发布】

JSP 表达式是 Web 服务器把脚本语言表达式的值转化成一个 String 对象，插入到返回给客户端的数据流中。表达式是在<%=和%>这两个标签之间定义的。

9. 隐含对象是什么意思？有哪些隐含对象？

JSP 隐含对象是页面中的一些 Java 对象，JSP 容器让这些 Java 对象可以为开发者所使用。开发者不用明确的声明就可以直接使用他们。JSP 隐含对象也叫做预定义变量。下面列出了 JSP 页面中的隐含对象：

application	page	request	response	session
exception	out	config	pageContext	

十三、框架技术

1. 谈谈 Hibernate 与 Ibatis 的区别，哪个性能会更高一些？

① Hibernate 偏向于对象的操作达到数据库相关操作的目的；而 ibatis 更偏向于 sql 语句的优化。

② Hibernate 使用的查询语句是自己的 hql，而 ibatis 则是标准的 sql 语句。

③ Hibernate 相对复杂，不易学习；ibatis 类似 sql 语句，简单易学。

性能方面：

① 如果系统数据处理量巨大，性能要求极为苛刻时，往往需要人工编写高性能的 sql 语句或存储过程，此时 ibatis 具有更好的可控性，因此性能优于 Hibernate。

② 同样的需求下，由于 hibernate 可以自动生成 hql 语句，而 ibatis 需要手动写 sql 语句，此时采用 Hibernate 的效率高于 ibatis。

2. 对 Spring 的理解，项目中都用什么？怎么用的？对 IOC 和 AOP 的理解及实现原理

Spring 是一个开源框架，处于 MVC 模式中的控制层，它能应对需求快速的变化，其主要原因是它有一种面向切面编程（AOP）的优势，其次它提升了系统性能，因为通过依赖注入机制（IOC），系统中用到的对象不是在系统加载时就全部实例化，而是在调用到这个类时才会实例化该类的对象，从而提升了系统性能。这两个优秀的性能使得 Spring 受到许多 J2EE 公司的青睐。

Spring 的**优点**：

①降低了组件之间的耦合性，实现了软件各层之间的解耦。

②可以使用容器提供的众多服务，如事务管理，消息服务，日志记录等。

③容器提供了 AOP 技术，利用它很容易实现如权限拦截、运行期监控等功能。

④Spring 中 AOP 技术是设计模式中的动态代理模式。只需实现 jdk 提供的动态代理接口 InvocationHandler，所有被代理对象的方法都由 InvocationHandler 接管实际的处理任务。面向切面编程中还要理解切入点、切面、通知、织入等概念。

⑤Spring 中 IOC 则利用了 Java 强大的反射机制来实现。所谓依赖注入即组件之间的依赖关系由容器在运行期决定。其中依赖注入的方法有两种，通过构造函数注入，通过 set 方法进行注入。

3. 描述 struts 的工作流程

① 在 web 应用启动时，加载并初始化 ActionServlet， ActionServlet 从 struts-config.xml 文件中读取配置信息，将它们存放到各个配置对象中。

② 当 ActionServlet 接收到一个客户请求时，首先检索和用户请求相匹配的 ActionMapping 实例，如果不存在，就返回用户请求路径无效信息。

③ 如果 ActionForm 实例不存在，就创建一个 ActionForm 对象，把客户提交的表单数据保存到 ActionForm 对象中。

④ 根据配置信息决定是否需要验证表单，如果需要，就调用 ActionForm 的 validate()方法，如果 ActionForm 的 validate()方法返回 null 或返回一个不包含 ActionMessage 的 ActionErrors 对象，就表示表单验证成功。

⑤ ActionServlet 根据 ActionMapping 实例包含的映射信息决定请求转发给哪个 Action，如果相应的 Action 实例不存在，就先创建一个实例，然后调用 Action 的 execute()方法。

⑥ Action 的 execute()方法返回一个 ActionForward 对象，ActionServlet 再把客户请求转发给 ActionForward 对象指向的 JSP 组件。

⑦ ActionForward 对象指向的 JSP 组件生成动态网页，返回给客户。

设计模式

总体来说设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

工厂模式：工厂模式是一种经常被使用到的模式，根据工厂模式实现的类可以根据提供的数据生成一组类中某一个类的实例，通常这一组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。首先需要定义一个基类，该类的子类通过不同的方法实现了基类中的方法。然后需要定义一个工厂类，工厂类可以根据条件生成不同的子类实例。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。

其实还有两类：并发型模式和线程池模式。

单例模式的要点

单例模式(singleton)顾名思义，就是只有一个实例。

单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。使用 Singleton 的好处在于可以节省内存，因为它限制了实例的个数，有利于 Java 垃圾回（garbage collection）

一是某个类只能有一个实例；

二是它必须自行创建这个实例；

三是它必须自行向整个系统提供这个实例

1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。

2、省去了 new 操作符，降低了系统内存的使用频率，减轻 GC 压力。

3、类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统就乱了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程。

【单例模式优缺点】

【优点】

①实例控制：单例模式会阻止其他对象实例化自己的单例对象的副本，从而确保所有对象都访问唯一实例。

②灵活性：因为类控制了实例化过程，所以类可以灵活更改实例化过程。

【缺点】

①**开销**：虽然数量很少，但如果每次对象请求引用时都要检查是否存在类的实例，将仍然需要一些开销。可以通过使用静态初始化解决此问题。

②**可能的开发混淆**：使用单例对象（尤其在类库中定义的对象）时，开发人员必须记住自己不能使用 new 关键字实例化对象。因为可能无法访问库源代码，因此应用程序开发人员可能会意外发现自己无法直接实例化此类。

③**对象生存期**：不能解决删除单个对象的问题。在提供内存管理的语言中（例如基于.NET Framework 的语言），只有单例类能够导致实例被取消分配，因为它包含对该实例的私有引用。在某些语言中（如 C++），其他类可以删除对象实例，但这样会导致单例类中出现悬浮引用。

写一个 Singleton 出来

下面是代码分析

```
//第一种：饱汉模式
/*
* 定义一个类，它的构造函数为 private 的，它有一个 static 的 private 的该类变量，在类初始化时实例化，通过一个 public 的 getInstance 方法获取对它的引用,继而调用其中的方法。
 */
public class SingleTon {
   private SingleTon(){ }
   //实例化放在静态代码块里可提高程序的执行效率，但也可能更占用空间
   private final static SingleTon instance =new SingleTon();
   //这里提供了一个供外部访问本 class 的静态方法，可以直接访问
   public static SingleTon getInstance(){
      return instance;
   }
}
```

注意：像这样毫无线程安全保护的类，如果我们把它放入多线程的环境下，肯定就会出现问题了，如何解决？我们首先会想到对 getInstance 方法加 synchronized 关键字，如下：

```
//第二种：饥汉模式
public class SingleTon {
 /* 持有私有静态实例，防止被引用，此处赋值为 null，目的是实现延迟加载 */
   private static SingleTon instance = null;
   public static synchronized SingleTon getInstance() {
   //这个方法比上面有所改进，不用每次都进行生成对象，只是第一次
   //使用时生成实例，提高了效率！
   if (instance == null)
      instance = new SingleTon();
      return instance;
   }
}
```

注意：但是，synchronized 关键字锁住的是这个对象，这样的用法，在性能上会有所下降，因为每次调用 getInstance()，都要对对象上锁，事实上，只有在第一次创建对象的时候需要加锁，之后就不需要了，所以，这个地方需要改进。我们改成下面这个：

```
public static Singleton getInstance() {
        if (instance == null) {
            synchronized (instance) {
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
        return instance;
    }
```

似乎解决了之前提到的问题，将 synchronized 关键字加在了内部，也就是说当调用的时候是不需要加锁的，只有在 instance 为 null，并创建对象的时候才需要加锁，性能有一定的提升。但是，这样的情况，还是有可能有问题的，看下面的情况：在 Java 指令中创 建对象和赋值操作是分开进行的，也就是说 instance = new Singleton();语句是分两步执行的。但是 JVM 并不保证这两个操作的先后顺序，也就是说有可能 JVM 会为新的 Singleton 实例分配空间，然后直接赋值给 instance 成员，然后再去初始化这个 Singleton 实例。这样就可能出错了，我们以 A、B 两个线程为例：

a>A、B 线程同时进入了第一个 if 判断

b>A 首先进入 synchronized 块，由于 instance 为 null，所以它执行 instance = new Singleton();

c>由于 JVM 内部的优化机制，JVM 先画出了一些分配给 Singleton 实例的空白内存，并赋值给 instance 成员（注意此时 JVM 没有开始初始化这个实例），然后 A 离开了 synchronized 块。

d>B 进入 synchronized 块，由于 instance 此时不是 null，因此它马上离开了 synchronized 块并将结果返回给调用该方法的程序。

e>此时 B 线程打算使用 Singleton 实例，却发现它没有被初始化，于是错误发生了。

所以程序还是有可能发生错误，其实程序在运行过程是很复杂的，从这点我们就可以看出，尤其是在写多线程环境下的程序更有难度，有挑战性。我们对该程序做进一步优化：

```
private static class SingletonFactory{
        private static Singleton instance = new Singleton();
    }
```

```
    public static Singleton getInstance(){
      return SingletonFactory.instance;
    }
```

实际情况是，单例模式使用内部类来维护单例的实现，JVM 内部的机制能够保证当一个类被加载的时候，这个类的加载过程是线程互斥的。这样当我们第一次调用 getInstance 的时候，JVM 能够帮我们保证 instance 只被创建一次，并且会保证把赋值给 instance 的内存初始化完毕，这样我们就不用担心上面的问题。同时该方法也只会在第一次调用的时候使用互斥机制，这样就解决了低性能问题。这样我们暂时总结一个完美的单例模式：

```
public class Singleton {
    /* 私有构造方法，防止被实例化 */
  private Singleton() {
  }
  /* 此处使用一个内部类来维护单例 */
  private static class SingletonFactory {
    private static Singleton instance = new Singleton();
  }
  /* 获取实例 */
  public static Singleton getInstance() {
    return SingletonFactory.instance;
  }
  /* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 */
  public Object readResolve() {
    return getInstance();
  }
}
```

其实说它完美，也不一定，如果在构造函数中抛出异常，实例将永远得不到创建，也会出错。所以说，十分完美的东西是没有的，我们只能根据实际情况，选择最适合自己应用场景的实现方法。也有人这样实现：因为我们只需要在创建类的时候进行同步，所以只要将创建和 getInstance()分开，单独为创建加 synchronized 关键字，也是可以的：

```
public class SingletonTest {
```

```
    private static SingletonTest instance = null;
    private SingletonTest() {
    }
    private static synchronized void syncInit() {
        if (instance == null) {
            instance = new SingletonTest();
        }
    }
    public static SingletonTest getInstance() {
        if (instance == null) {
            syncInit();
        }
        return instance;
    }
}
```

考虑性能的话，整个程序只需创建一次实例，所以性能也不会有什么影响。

排序算法

名称	数据对象	稳定性	时间复杂度		空间复杂度	描述
			平均	最坏		
插入排序	数组、链表	✓	$O(n^2)$		$O(1)$	(有序区,无序区)。把无序区的第一个元素插入到有序区的合适的位置。对数组：比较得少，换得多。
直接选择排序	数组	✗	$O(n^2)$		$O(1)$	(有序区,无序区)。在无序区里找一个最小的元素跟在有序区的后面。 对数组：比较得多，换得少。
	链表	✓				
堆排序	数组	✗	$O(nlogn)$		$O(1)$	(最大堆,有序区)。从堆顶把根卸出来放在有序区之前，再恢复堆。
归并排序	数组、链表	✓	$O(nlogn)$		$O(n)+O(logn)$, 如果不是从下到上	把数据分为两段，从两段中逐个选最小的元素移入新数据段的末尾。可从上到下或从下到上进行。
快速排序	数组	✗	$O(nlogn)$	$O(n^2)$	$O(logn),O(n)$	(小数,枢纽元,大数)。
Accum qsort	链表	✓	$O(nlogn)$	$O(n^2)$	$O(logn),O(n)$	(无序区,有序区)。把无序区分为(小数,枢纽元,大数)，从后到前压入有序区。
决策树排序		✓	$O(logn!)$		$O(n!)$	O(n) <O(logn!) <O(nlogn)
计数排序	数组、链表	✓	$O(n)$		$O(n+m)$	统计小于等于该元素值的元素的个数 i，于是该元素就放在目标数组的索引 i位。(i≥0)
桶排序	数组、链表	✓	$O(n)$		$O(m)$	将值为 i 的元素放入i 号桶，最后依次把桶里的元素倒出来。
基数排序	数组、链表	✓	$O(k*n)$,最坏: $O(n^2)$			一种多关键字的排序算法，可用桶排序实现。

冒泡排序经过优化以后，最好时间复杂度可以达到 O(n)。设置一个标志位，如果有一趟比较中没有发生任何交换，可提前结束，因此在正序情况下，时间复杂度 为 O(n)。

选择排序在最坏和最好情况下，都必须在剩余的序列中选择最小（大）的数，与已排好序的序列后一个位置元素做交换，依次最好和最坏时间复杂度均为 O(n^2)。

插入排序是在把已排好序的序列的后一个元素插入到前面已排好序(需要选择合适的位置)的序列中，在正序情况下时间复杂度为 O(n)。

堆是完全二叉树，因为树的深度一定是 log(n)+1，最好和最坏时间复杂度均为 O(n*log(n))。归并排序是将大数组分为两个小数组，依次递归，相当 于二叉树，深度为 log(n)+1，因此最好和最坏时间复杂度都是 O(n*log(n))。

快速排序在正序或逆序情况下，每次划分只得到比上一次划分少一 个记录的子序列，用递归树画出来，是一棵斜树，此时需要 n-1 次递归，且第 i 次划分要经过 n-i 次关键字比较才能找到第 i 个记录，因此时间复杂度 是\sum_{i=1}^{n-1}(n-i)=n(n-1)/2，即 O(n^2)。

冒泡排序

```
 5  * description:冒泡排序
 6  * @author HZL
 7  * @date 2016年1月20日 下午4:16:13
 8  *
 9  */
10 public class BubbleSort {
11
12     static int[] data = {9,2,7,19,100,98,63,208,55,78};
13
14     public static void bubbleSort(){          //从后往前比较，小的数字依次往前冒泡，直到排序完成
15         int tmp = 0;
16
17         for(int i=0;i<data.length;i++){       //需要遍历n趟，每一趟冒泡一个最小的数字
18             for(int j=data.length-1;j>i;j--){
19                 if(data[j] < data[j-1]){      //在每一趟中，从后往前相邻进行比较，小的数字往前冒泡，直到该趟最前
20                     tmp = data[j];
21                     data[j] = data[j-1];
22                     data[j-1] = tmp;
23                 }
24             }
25         }
26     }
27
28     public static void print(){
29         for(int i =0;i<data.length;i++){
30             System.out.print(data[i]+" ");
31         }
32     }
33
34     public static void main(String[] args){
35         BubbleSort.print();
36         System.out.println();
37         System.out.println("==============");
38         BubbleSort.bubbleSort();
39         BubbleSort.print();
40     }
```

选择排序

```
 * description:选择排序
 * @author HZL
 * @date 2016年1月20日 下午6:23:07
 *
 */
public class SelectSort {

    static int[] data = {9,2,7,19,100,98,63,208,55,78};

    public static void selectSort(){        //按位置排序，第1个位置该排谁，第2个位置该排谁。。。。。
        int k =0 ,tmp =0;
        for(int i=0;i<data.length;i++){     //按位置排序，首先找到最小值，排在第1位，然后在待排序的数中选择最小的值，排在第2位。。。。。。
            k=i;
            for(int j=i+1; j<data.length;j++){
                if(data[j] < data[k]){
                    k = j;                  //k:用于保存每次的最小值的下标，然后和i位置的数字进行交换
                }
            }
            tmp = data[k];
            data[k] = data[i];
            data[i] = tmp;
        }
    }

    public static void print(){
        for(int i=0;i<data.length;i++){
            System.out.print(data[i]+" ");
        }
    }

    public static void main(String[] args){
        SelectSort.print();
        System.out.println();
        System.out.println("========================");
        SelectSort.selectSort();
        SelectSort.print();
    }
}
```

插入排序

```
 * description:插入排序
 * @author HZL
 * @date 2016年1月20日 下午6:25:19
 *
 */
public class InsertSort {

    static int[] data = {9,2,7,19,100,98,63,208,55,78};

    public static void insertSort(){
        int tmp=0,j=0;
        for(int i=1;i<data.length;i++){
            tmp = data[i];                  //把data[i]保存起来，tmp用来保存即将进行排序的数字
            j = i-1;                        //j是i的前一个数字的下标
            while(j>=0 &&tmp<data[j]){
                data[j+1] = data[j];        //每次跟前面的数字进行比较，如果前面的数字大就把该数字向后移动1位
                j--;
            }
            data[j+1] = tmp;                //j+1就是tmp应该插入的位置，直接插入即可
        }
    }
```

```
26
27      public static void print(){
28          for(int i=0;i<data.length;i++){
29              System.out.print(data[i]+" ");
30          }
31      }
32
33      public static void main(String[] args){
34          InsertSort.print();
35          System.out.println();
36          System.out.println("====================");
37          InsertSort.insertSort();
38          InsertSort.print();
39      }
40  }
41
```

快速排序

```
 5   * description:快速排序
 6   * @author HZL
 7   * @date 2016年1月20日 下午7:05:34
 8   *
 9   */
10  public class QuickSort {
11
12      static int[] data = {9,2,7,19,100,98,63,208,55,78};
13
14      public int partition(int[] data,int low,int high){  //选取第一个数为key,把数组分为两部分
15          int key = data[low];                            //先把第一个数保存在变量key中，最后放到合适的位置上
16          while(low < high){
17              while(low < high && data[high]>=key)        //如果data[high]比key大，就跳过high，再往前找一个数
18                  high--;                                 //直到data[high]比key小，然后进行交换
19                  data[low] = data[high];
20
21              while(low < high && data[low]<=key)         //同理......
22                  low++;
23                  data[high] = data[low];
24          }
25          data[low] = key;                                //把事前保存的key放到合适的位置low上
26          return low;                                     //返回这个合适的位置，就是可以使数字分居左右两侧的key位置
```

```
29      public int[] sort(int low, int high){               //递归调用sort,直到low == high，递归结束，排序完成
30          if(low < high){
31              int result = partition(data, low, high);
32              sort(low,result-1);
33              sort(result+1,high);
34          }
35          return data;
36      }
37      public void print(int[] data){
38          for(int i = 0; i<data.length; i++){
39              System.out.print(data[i]+" ");
40          }
41      }
42      public static void main(String[] args){
43          QuickSort qs = new QuickSort();
44          qs.data = data;
45          qs.print(data);
46          System.out.println();
47          System.out.println("====================");
48          qs.sort(0, data.length-1);
49          qs.print(data);
50      }
```

归并排序

```
 * description:归并排序
 * @author HZL
 * @date 2016年1月20日 下午7:36:09
 *
 */
public class MergeSort {

    public static void mergeSort(int[] a){
        Sort(a,0,a.length-1);       //下标的范围：[0 ~ a.length-1]
    }

    private static void Sort(int[] a, int left, int right) {
        if(left < right){               //分解条件：left < right ，当left >= right时，说明已经分解到单个数字了
            int mid = (left + right)/2; //选取中间位置
            Sort(a,left,mid);           //递归对左半部分进行分解，直到不满足(left < right)条件为止
            Sort(a,mid+1,right);        //递归对右半部分进行分解，直到不满足(left < right)条件为止
            Merge(a,left,mid,right);    //分解过程完成，进行合并
        }

    }

    public static void Merge(int[] a, int left, int mid, int right) {    //合并算法，就当做是两个数组进行合并
        int[] tmp = new int[a.length];                                    //附件空间tmp数组，长度和原数组长度一致
        int i = left;
        int j = mid+1;
        int k = 0;
        while(i <= mid && j<=right){
            if(a[i] <= a[j]){
                tmp[k++] = a[i++];
            }else{
                tmp[k++] = a[j++];
            }
        }
        while(i<=mid){
            tmp[k++] = a[i++];
        }
        while(j<=right){
            tmp[k++] = a[j++];
        }
        for(i=0;i<k;i++){           //k在上面多加了一次，所以此时，i<k 而不是 i<= k [注意！]
            a[left+i] = tmp[i];     //合并完成之后，把此时已经有序的附加数组tmp拷贝回原数组a[]
        }
    }
    public static void print(int[] a){
        for(int i=0;i<a.length;i++){
            System.out.print(a[i]+" ");
        }
    }
    public static void main(String[] args){
        int[] a = {8,16,99,732,10,1,29,66};
        MergeSort.print(a);
        System.out.println();
        System.out.println("==========================");
        MergeSort.mergeSort(a);
        MergeSort.print(a);
    }

}
```

二分查找

```
 * description:二分查找
 * @author HZL
 * @date 2016年1月20日 下午7:57:42
 *
 */
public class BinarySearch {

    static int[] data={2,7,9,19,55,63,78,97,100,208};

    public static int binarySearch(int key,int low,int high){    //传入3个参数，关键字：key 查找范围：low和high
        if(low <= high){                                         //递归调用条件，不满足这个条件时查找结束
            int mid = (high + low)/2;
            if(data[mid] == key){                                //命中，则返回关键字下标
                return mid;
            }else if(data[mid] < key){
                return binarySearch(key, mid+1, high);           //向右半部分进行，继续调用递归返回查找过程
            }else if(data[mid] > key){
                return binarySearch(key,low,mid-1);              //向左半部分进行，继续调用递归返回查找过程
            }
        }
        return -1;                                               //查找过程结束，没有命中关键字，返回失败符号：-1
    }

    public static void main(String[] args){
        int result = BinarySearch.binarySearch(5, 0, data.length-1);
        System.out.println(result);
    }
}
```

字符串全排列

```
 * description:字符串的全排列
 * @author HZL
 * @date 2016年1月20日 下午8:42:02
 *
 */
public class Permutation {

    public static void permutation(char[] s,int from, int to){  //from：数组的第一个数的下标
        if(to <= 1)
            return ;
        if(from == to){             //递归结束条件：如果from == to,则代表最后一个数都已经递归了
            System.out.print(s);    //所以递归结束可以打印输出了
            System.out.println();
        }else{
            for(int i=from; i<=to; i++){     //i依次增加向后遍历，可以使得第一个数字依次和后面的每个数字交换
                swap(s,i,from);
                permutation(s, from+1, to); //递归调用  第二个数与后面的每个数进行交换
                swap(s,from,i);             //递归完成后，本轮遍历结束，把数字交换回原数组，进行下一轮交换和递归
            }
        }
    }

    public static void swap(char[] s,int i,int j){
        char tmp = s[i];
        s[i] = s[j];
        s[j] = tmp;
    }

    public static void main(String[] args){
        char[] s = {'a','b','c'};
        Permutation.permutation(s, 0, s.length-1);
    }
}
```

字符串全组合

```
 * description:字符串的全组合
 * @author HZL
 * @date 2016年1月20日 下午8:53:08
 *
 */
public class Combination {
    public static void combination(char[] s){
        int len_j = s.length;                           //len_j：指针j需要循环遍历的次数，就是数组本身的长度
        int n_i = 1<<len_j;                             //n_i：指针i需要循环遍历的次数，[1,2的n次方]
        for(int i=1; i<n_i;i++){                        //i的取值：001 010 011 100 101 110 111
            StringBuffer sb = new StringBuffer();       //每一趟i 都对应于该趟应该输出的字符
            for(int j=0; j<len_j;j++){                  //1<<j：001 010 100
                if((i&(1<<j))!=0){                      //指针i和指针j进行交叉对照，所有需要用append进行拼接，从而得出该趟所有的字符
                    sb.append(s[j]);
                }
            }
            System.out.print(sb+" ");                   //这一趟遍历完毕，把字符打印出来，然后外层for循环进行下一趟打印
        }
    }

    public static void main(String[] args){
        char[] s={'a','b','c'};
        Combination.combination(s);
    }
}
```

字符串匹配

```
 * description:字符串匹配
 * @author HZL
 * @date 2016年1月20日 下午9:03:38
 *
 */
public class StringFind {

    public static void stringFind(String s1,String s2){
        char[] input = s1.toCharArray();
        char[] pattern = s2.toCharArray();

        if(s1 == null || s2==null)
            return ;

        for(int i=0;i<input.length;i++){
            if(input[i] == pattern[0] || pattern[0]=='?'){  //input[i]匹配成功，则从i开始往后匹配
                int count = 1;                              //count记录匹配的次数
                int j = 1;                                  //j为pattern数组的下标，0已匹配成功，所以从1开始
                while(j<pattern.length){
                    if(pattern[j] == '?'){                  //pattern[j]=?,则视为该字符串匹配成功，进行下一个字符匹配
                        i++;
                        j++;
                        count++;
                    }else if(pattern[j] !='?' && input[++i] == pattern[j]){

                        j++;
                        count++;
                    }else{
                        break;              //当某个字符不匹配时，直接跳出while循环
                    }
                }
                if(count == pattern.length){ //匹配过程结束，验证是否全部匹配成功
                    for(int m=i-count+1;m<i-count+1+pattern.length;m++){
                        System.out.print(input[m]); //成功，把所有匹配的字符打印出来
                    }
                }else{                          //没有全部匹配成功，则恢复到第一次匹配成功的下标处
                    i = i-count+1;              //准备下一轮匹配
                }
            }
        }
    }

    public static void main(String[] args){
        String s1 = "abcdef";
        String s2 = "b?de";
        StringFind.stringFind(s1,s2);
    }

}
```

字符串移动

```
 * description:字符串移动
 * @author HZL
 * @date 2016年1月20日 下午9:24:31
 *
 */
public class StringMove {

    public static void stringMove(String s){
        if(s ==null)
            return ;
        char[] input=s.toCharArray();
        int count1=0, count2=0;      //count1记录原字符串中字符的个数，count2记录原字符串中数字的个数

        for(int i=0;i<input.length;i++){
            if(Character.isDigit(input[i])){
                count2++;
            }else{
                count1++;
            }
        }

        char[] s1 = new char[count1];    //把原字符串截取成两部分，字符的部分放在s1中
        char[] s2 = new char[count2];    //数字部分放在s2中

        int j=0,k=0,m=0;

        for(int i=0;i<input.length;i++){
            if(Character.isDigit(input[i])){
                s2[j++] = input[i];
            }else{
                s1[k++] = input[i];
            }
        }
        String s3 = new String(s1);      //把s1数组转换成字符串
        String s4 = new String(s2);      //把s2数组转换成字符串

        System.out.println(s3+s4);       //输出整个字符串
    }

    public static void main(String[] args){
        String s="abcd345ef89";
        StringMove.stringMove(s);
    }

}
```