- 为什么需要 public static void main(String[] args)这个方法？

public static void main(String[] args)为 Java 程序的入口方法，JVM 在运行程序时，首先会查找 main()方法。其中，public 是权限修饰符，表明任何类或对象都可以访问这个方法，static 表明 main()方法是一个静态方法，即方法中的代码是存储在静态存储区的，只要类被加载后，就可以使用该方法而不需要通过实例化对象来访问，可以直接通过类名.main()直接访问，JVM 在启动时就是按照上述方法的签名（必须有 public 与 static 修饰，返回值为 void，且方法的参数为字符串数组）来查找方法的入口地址，若能找到，就执行；找不到，则会报错。void 表明方法没有返回值，main 是 JVM 识别的特殊方法名，是程序的入口方法。字符串数组参数 args 为开发人员在命令行状态下与程序交互提供了一种手段。

因为 main 是程序的入口方法，所以当程序运行时，第一个执行的方法就是 main()方法。通常来讲，要执行一个类的方法，先必须实例化一个类的对象，然后通过对象来调用这个方法。但**由于 main 是程序的入口方法，此时还没有实例化对象，因此在编写 main()方法时就要求不需要实例化对象就可以调用这个方法，鉴于此，main()方法需要被定义成 public 与 static**。

- main()方法是否还有其他可用的定义格式？

（1）public 与 static 没有先后顺序关系。

（2）也可以把 main()方法定义为 final。

（3）也可以用 synchronized 来修饰 main()方法。

- Java 程序初始化的顺序是怎样的？

Java 程序的初始化一般遵循 3 个原则（优先级依次递减）：(1) 静态对象（变量）优先于非静态对象（变量）初始化，其中，静态对象（变量）只初始化一次，而非静态对象（变量）可能会初始化多次。(2) 父类优先于子类进行初始化。(3) 按照成员变量的定义顺序进行初始化。即使成员变量定义散布于方法定义之中，它们依然在任何方法（包括构造函数）被调用之前先初始化。

Java 程序初始化工作可以在许多不同的代码块中来完成，它们执行的顺序如下：**父类静态变量、父类静态代码块、子类静态变量、子类静态代码块、父类非静态变量、父类非静态代码块、父类构造函数、子类非静态变量、子类非静态代码块、子类构造函数**。

- 什么是构造函数？

构造函数是一种特殊的函数，用来在对象实例化时初始化对象的成员变量。在 Java 语言中，构造函数具有以下特点。

（1）构造函数必须与类的名字相同，并且**不能有返回值**（返回值也不能为 void）。

（2）每个类可以有多个构造函数。**当开发人员没有提供构造函数时，编译器在把源代码编译成字节码的过程中会提供一个没有参数默认的构造函数**，但该构造函数不会执行任何代码。如果开发人员提供了构造函数，那么编译器就不会再创建默认的构造函数了。

（3）构造函数可以有 0 个、1 个或 1 个以上的参数。

（4）构造函数总是伴随着 new 操作一起调用，且**不能由程序的编写者直接调用，必须要由系统调用**。构造函数在对象实例化时会被自动调用，且只运行一次；而普通的方法是在程序执行到它时被调用，且可以被该对象调用多次。

（5）构造函数的主要作用是完成对象的初始化工作。

（6）构造函数**不能被继承**，因此，它**不能被覆盖**，但是构造函数能够被重载，可以使用不同的参数个数或参数类型来定义多个构造函数。

（7）子类可以通过 super 关键字来显式地调用父类的构造函数，**当父类没有提供无参数的构造函数时，子类的构造函数中必须显式地调用父类的构造函数**。如果父类提供了无参数的构造函数，此时子类的构造函数就可以不显式地调用父类的构造函数，**在这种情况下编译器会默认调用父类提供的无参数的构造函数**。当有父类时，在实例化对象时会先执行父类的构造函数，然后执行子类的构造函数。

（8）当父类和子类都没有定义构造函数时，编译器会为父类生成一个默认的无参数的构造函数，给子类也生成一个默认的无参数的构造函数。此外，默认构造器的修饰符只跟当前类的修饰符有关。

（9）**普通方法可以与构造函数有相同的方法名**。

接口中成员的作用域修饰符都是 public，接口中的常量值默认使用 public static final 修饰。

- 浅复制和深复制有什么区别？

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

深复制：被复制对象的所有变量都含有与原来对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将指向被复制的新对象，而不再是原有的那些被引用的对象。换言之，深复制把复制的对象所引用的对象都复制了一遍。

- 什么是反射机制？

反射机制是 Java 语言中一个非常重要的特性，它允许程序在运行时进行自我检查，同时也允许对其内部的成员进行操作。由于反射机制能够实现在运行时对类进行装载，因此能够增加程序的灵活性，但是不恰当地使用反射机制，也会严重影响系统的性能。

在反射机制中，Class 是一个非常重要的类，那么如何才能获取 Class 类呢？

（1）Class.forName（“类的路径”）

（2）类名.Class

（3）实例.getClass()

- Java 创建对象的方式有几种？

（1）通过 **new** 语句实例化一个对象。

（2）通过**反射**机制创建对象。

（3）通过 **clone()**方法创建一个对象。

（4）通过**反序列化**的方式创建对象。

- 面向对象有哪些特征？

（1）**抽象**。抽象就是忽略一个主题中与当前目标无关的那些方面，以便更充分地注意与当前目标有关的方面。抽象并不打算了解全部问题，而只是选择其中的一部分，暂时不用部分细节。抽象包括两个方面：一是过程抽象；二是数据抽象。

（2）**继承**。继承是一种联结类的层次结构，并且允许和鼓励类的重用，它提供了一种明确表述共性的方法。

（3）**封装**。封装是指将客观事物抽象成类，每个类对自身的数据和方法实行保护。类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。

（4）**多态**。多态是指允许不同的类的对象对同一消息作出响应。

- 什么是继承？

继承主要有如下几个特性：

（1）Java 语言不支持多重继承，子类至多只能有一个父类，但是可以通过实现多个接口来达到多重继承的目的。

（2）子类只能继承父类的非私有（public 与 protected）成员变量与方法。

（3）**当子类中定义的成员变量和父类中定义的成员变量同名时，子类中的成员变量会覆盖父类的成员变量，而不会继承**。

（4）当子类中的方法与父类中的方法有相同的函数签名（相同的方法名，相同的参数个数与类型）时，子类将会覆盖父类的方法，而不会继承。

- 重载和覆盖有什么区别？

重载（overload）和覆盖（override）是 Java 多态性的不同表现方式。其中，重载是在一个类中多态性的一种表现，是指在一个类中定义了多个同名的方法，它们或有不同的参数个数或有不同的参数类型。在使用重载时，需要注意以下几点：

（1）重载是通过不同的方法参数来区分的，例如不同的参数个数、不同的参数类型或不同的参数顺序。

（2）不能通过方法的访问权限、返回值类型和抛出的异常类型来进行重载。

（3）**对于继承来说，如果基类方法的访问权限为 private，那么就不能在派生类对其重载；如果派生类也定义了一个同名的函数，这只是一个新的方法，不会达到重载的效果**。

覆盖是指派生类函数覆盖基类函数。覆盖一个方法并对其重写，以达到不同的作用。在使用覆盖时需要注意以下几点：

（1）派生类中的覆盖方法必须要和基类中被覆盖的方法有相同的函数名和参数。

（2）派生类中的覆盖方法的返回值必须和基类中被覆盖方法的**返回值相同**。

（3）派生类中的覆盖方法所抛出的异常必须和基类（或其子类）中被覆盖的方法所抛出的异常一致。

（4）**基类中被覆盖的方法不能为 private，否则其子类只是定义了一个方法，并没有对其覆盖**。

重载与覆盖的区别主要有以下几个方面：

（1）覆盖是子类和父类之间的关系，是垂直关系；重载是同一个类中方法之间的关系，是水平关系。

（2）覆盖只能由一个方法或只能由一对方法产生关系；重载是多个方法之间的关系。

（3）覆盖要求参数列表相同；重载要求参数列表不同。

（4）覆盖关系中，调用方法体是根据对象的类型（对象对应存储空间类型）来决定；而重载关系是根据调用时的实参表与形参表来选择方法体的。

- 抽象类与接口有什么异同？

接口与抽象类的相同点如下：

（1）都不能被实例化。

（2）接口的实现类或抽象类的子类都只有实现了接口或抽象类中的方法后才能被实例化。

接口与抽象类的不同点如下：

（1）接口只有定义，其方法不能在接口中实现，只有实现接口的类才能实现接口中定义的方法，而抽象类可以有定义与实现，即其方法可以在抽象类中被实现。

（2）接口需要实现（用 implements），但抽象类只能被继承（用 extends）。一个类可以实现多个接口，但一个类只能继承一个抽象类，因此使用接口可以间接地达到多重继承的目的。

（3）接口强调特定功能的实现，其设计理念是“has-a”关系；而抽象类强调所属关系，其设计理念为“is-a”关系。

（4）接口中定义的成员变量默认为 public static final，只能够有静态的而不能被修改的数据成员，而且**必须给其赋初值**，其**所有成员方法都是 public、abstract 的，而且只能被这两个关键字修饰**。而抽象类可以有自己的数据成员变量，也可以有非抽象的成员方法，而且，抽象类中的成员变量默认为 default（本包可见），当然也可以定义为 private、protected 和 public，这些成员变量可以在子类中被重新定义，也可以被重新赋值，**抽象类中的抽象方法（其前有 abstract 修饰）不能用 private、static、synchronized、native 等访问修饰符修饰**，同时方法必须以分号结尾，并且不带花括号。所以，当功能需要累积时，用抽象类；不需要累积时，用接口。

（5）接口被运用于实现比较常用的功能，便于日后维护或者添加删除方法；而抽象类更倾向于充当公共类的角色，不适用于日后重新对里面的代码进行修改。

接口可以继承接口，抽象类可以实现接口，抽象类也可以继承具体类。**抽象类也可以有 main 方法**。

- 内部类有哪些？

静态内部类是指声明为 static 的内部类，它可以**不依赖于外部类实例而被实例化**，而通常的内部类需要在外部类实例化后才能实例化。静态内部类不能与外部类有相同的名字，**不能访问外部类的普通成员变量，只能访问外部类中的静态成员和静态方法（包括私有类型）**。

成员内部类为非静态内部类，它可以自由地引用外部类的属性和方法，无论这些属性和方法是静态的还是非静态的。但是它与一个实例绑定在了一起，不可以定义静态的属性和方法。**只有在外部的类被实例化后，这个内部类才能被实例化。非静态内部类不能有静态成员**。

局部内部类指的是定义在一个代码块内的类，它的作用范围为其所在的代码块，是内部类中最少使用到的一种类型。局部内部类像局部变量一样，不能被 public、protected、private 以及 static 修饰，只能访问方法中定义为 final 类型的局部变量。

匿名内部类是一种没有类名的内部类，不使用关键字 class、extends、implements，没有构造函数，它必须继承其他类或实现其他接口。在使用匿名内部类时，需要牢记以下几个原则：

（1）**匿名内部类不能有构造函数**。

（2）**匿名内部类不能定义静态成员、方法和类**。

（3）**匿名内部类不能是 public、protected、private、static**。

（4）只能创建匿名内部类的一个实例。

（5）一个匿名内部类一定是在 new 的后面，这个匿名内部类必须继承一个父类或实现一个接口。

（6）因为匿名内部类为局部内部类，所以局部内部类的所有限制都对其生效。

Java 语言中任何类都继承自 Object 类，getClass()方法在 Object 类中被定义为 final 与 native，子类不能覆盖该方法。因此 **this.getClass()和 super.getClass()最终都调用的是 Object 中的 getClass()方法**。而 Object 的 getClass()方法的释义是：**返回此 Object 的运行时类**。

当子类构造函数需要显式调用父类构造函数时，super()必须为构造函数中的第一条语句。

- final、finally 和 finalize 有什么区别？

（1）final 用于声明属性、方法和类，分别表示属性不可变、方法不可覆盖和类不可被继承（不能再派生出新的子类）。

final 属性：被 final 修饰的变量不可变（引用不可变）。被 final 修饰的变量必须被初始化。一般可以通过以下几种方式对其进行初始化：**①在定义的时候初始化。②final 成员变量可以在初始化块中初始化，但不可在静态初始化块中初始化。③静态 final 成员变量可以在静态初始化块中初始化，但不可在初始化块中初始化。④在类的构造器中初始化，但静态 final 成员变量不可以在构造函数中初始化。**

final 方法：当一个方法声明为 final 时，该方法不允许任何子类重写这个方法，但子类仍然可以使用这个方法。另外，还有一种被称为 inline（内联）的机制，当调用一个被声明为 final 的方法时，直接将方法主体插入到调用处，而不是进行方法调用，这样做能提高程序效率。

final 参数：用来表示这个参数在函数内部不允许被修改。

final 类：当一个类被声明为 final 时，此类不能被继承，所有方法都不能被重写。但这并不表示 final 类的成员变量也是不可改变的，要学习做到 final 类的成员变量不可改变，必须给成员变量增加 final 修饰。一个类不能既被声明为 abstract，又被声明为 final。

（2）finally 作为异常处理的一部分，它只能用在 try/catch 语句中，并且附带一个语句块，表示这段语句最终一定被执行，经常被用在需要释放资源的情况下。

（3）finalize 是 Object 类的一个方法，在垃圾回收器执行时会调用被回收对象的 finalize()方法，可以覆盖此方法来实现对其他资源的回收，例如关闭文件等。**一旦垃圾回收器准备好释放对象占用的空间，将首先调用其 finalize()方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占用的内存。**

- assert 有什么作用？

断言（assert）是一种软件调试方法。在实际开发中，assert 主要用来保证程序的正确性，通常在程序开发和测试时使用。为了提高程序运行的效率，在软件发布后，assert 检查默认是关闭的。

用 java –ea Test 来开启 assert 检查。

assert 的应用范围很多，主要包括：（1）检查控制流；（2）检查输入参数是

否有效；(3) 检查函数结果是否有效；(4) 检查程序不变量。虽然 assert 的功能与 if 判断类似，但二者存在着本质的区别：**assert 一般在调试程序时使用，但如果不小心用 assert 来控制了程序的业务流程，那在调试结束后去掉 assert 就意味着修改了程序的正常逻辑，这样的做法是非常危险的；而 if 判断是逻辑判断，本身就是以用以控制程序流程的**。

- static 关键字有哪些作用？

static 关键字主要有两种作用：第一，为某特定数据类型或对象分配单一的存储空间，而与创建对象的个数无关。第二，实现某个方法或属性与类而不是对象关联在一起，也就是说，在不创建对象的情况下就可以通过类来直接调用方法或使用类的属性。

（1）static 成员变量

静态变量属于类，在内存中只有一个复制（所有实例都指向同一个内存地址），只要静态变量所在的类被加载，这个静态变量就会被分配空间。

（2）static 成员方法

static 方法是类的方法，不需要创建对象就可以被调用，而非 static 方法是对象的方法，只有对象被创建出来以后才可以被使用。

static 方法中不能使用 this 和 super 关键字，不能调用非 static 方法，只能访问所属类的静态成员变量和成员方法，因为当 static 方法被调用时，这个类的对象可能还没被创建，即使已经被创建了，也无法确定调用哪个对象的方法。同理，**static 方法也不能访问非 static 类型的变量**。

（3）static 代码块

static 代码块在类中是独立于成员变量和成员函数的代码块的。它不在任何一个方法体内，**JVM 在加载类时会执行 static 代码块，如果有多个 static 代码块，JVM 将会按顺序来执行。static 代码块经常被用来初始化静态变量**。

（4）static 内部类

- 使用 switch 时有哪些注意事项？

switch 语句用于多分支选择，在使用 switch(expr)时，expr 只能是一个**枚举常量**（内部也是由整型或字符类型实现）或一个**整数表达式**，其中整数表达式可以是基本类型 int 或其对应的包装类 Integer，当然也包括不同的长度整型，例如 short。由于 byte、short 和 char 类型的值都能够被隐式地转换为 int 类型，因此这些类型以及它们对应的包装类型都可以作为 switch 的表达式。但是，long、

float、double、String 类型不能够隐式地转换为 int 类型，因此它们不能被用作 switch 的表达式。

随着 Java 语言的发展，在 Java 7 中，switch 开始支持 String 类型了。

从本质上讲，switch 对字符串的支持，其实是 int 类型值的匹配。它的实现原理如下：通过对 case 后面的 String 对象调用 hashCode()方法，得到一个 int 类型的 hash 值，然后用这个 hash 值来唯一标识这个 case。那么当匹配时，首先调用这个字符串 **hashCode()函数**，获取一个 hash 值（int 类型），用这个 hash 值来匹配所有 case，如果没有匹配成功，说明不存在；如果匹配成功了，接着会调用字符串的 **String.equals()方法进行匹配**。由此可以看出，**String 变量不能为 null**，同时，**switch 的 case 子句中使用的字符串也不能为 null**。

- strictfp 有什么作用？

关键字 strictfp 是 strict float point 的缩写，指的是精确浮点，它用来确保浮点数运算的准确性。**JVM 在执行浮点数运算时，如果没有指定 strictfp 关键字，此时计算结果可能会不精确，而且计算结果在不同平台或厂商的虚拟机上会有不同的结果，导致意想不到的错误**。而一旦使用了 strictfp 来声明一个类、接口或者方法，那么在所声明的范围内，Java 编译器以及运行环境会完全依照 IEEE 二进制浮点数算术标准（IEEE 754）来执行，在这个关键字声明的范围内所有浮点数的计算都是精确的。需要注意的是，当一个类被 strictfp 修饰时，所有方法都会自动被 strictfp 修饰。因此，**strictfp 可以保证浮点数运算的精确性，而且在不同的硬件平台上会有一致的运算结果**。

在 Java 语言中，**默认声明的小数是 double 类型的**，因此**在对 float 类型的变量进行初始化时需要进行类型转换**。

- 什么是不可变类？

在 Java 类库中，所有基本类型的包装类都是不可变类。String 也是不可变类。通常来讲，创建一个不可变类需要遵循下面 4 条基本原则：

（1）类中所有成员变量被 private 修饰。

（2）类中没有写或者修改成员变量的方法，例如 setxxx，只提供构造函数，一次生成，永不改变。

（3）确保类中所有方法不会被子类覆盖，可以通过把类定义为 final 或者把类中的方法定义为 final 来达到这个目的。

（4）**如果一个类成员不是不可变量，那么在成员初始化或者使用 get 方法获取该成员变量时，需要通过 clone 方法来确保类的不变性**。

（5）如果有必要，可覆盖 Object 类的 equals()方法和 hashCode()方法。在 equals()方法中，根据对象的属性值来比较两个对象是否相等，并且保证用 equals()方法判断为相等的两个对象的 hashCode()方法的返回值也相等，这可以保证这些对象能被正确地放到 HashMap 或 HashSet 集合中。

由于类的不可变性，在创建对象时就需要初始化所有成员变量，因此最好**提供一个带参数的构造函数来初始化这些成员变量**。

- 字符串创建与存储的机制是什么？

在 Java 语言中，字符串起着非常重要的作用，字符串的声明与初始化主要有如下两种情况：

（1）对于 String s1 = new String(“abc”)语句与 String s2 = new String(“abc”)语句，存在两个引用对象 s1、s2，两个内容相同的字符串对象“abc”，它们在内存中的地址是不同的。只要用 new 总会生成新的对象。

（2）对于 String s1 = “abc”语句与 String s2 = “abc”语句，在 JVM 中存在着一个字符串池，其中保存着很多 String 对象，并且可以被共享使用，s1、s2 引用的是同一个常量池中的对象。由于 String 的实现采用了 Flyweight 的设计模式，当创建一个字符串常量时，例如 String s= “abc”，会首先在字符串常量池中查找是否已经有相同的字符串被定义，其判断依据是 String 类 equals(Object obj)方法的返回值。若已经定义，则直接获取对其的引用，此时不需要创建新的对象；若没有定义，则首先创建这个对象，然后把它加入到字符串池中，再将它的引用返回。由于 String 是不可变类，一旦创建好了就不能被修改，因此 String 对象可以被共享而且不会导致程序的混乱。

具体而言：

String s = “abc”; //把“abc”放到常量池中，在编译时产生

String s = “ab” + “c”; //把“ab” + “c”转换为字符串常量“abc”放到常量区中

String s = new String(“abc”); //在运行时把“abc”放到堆里面

String s1 = “abc”; //在常量区里面存放了一个新的字符串对象

String s2 = “abc”; //s2 引用常量区中的对象，因此不会创建新的对象

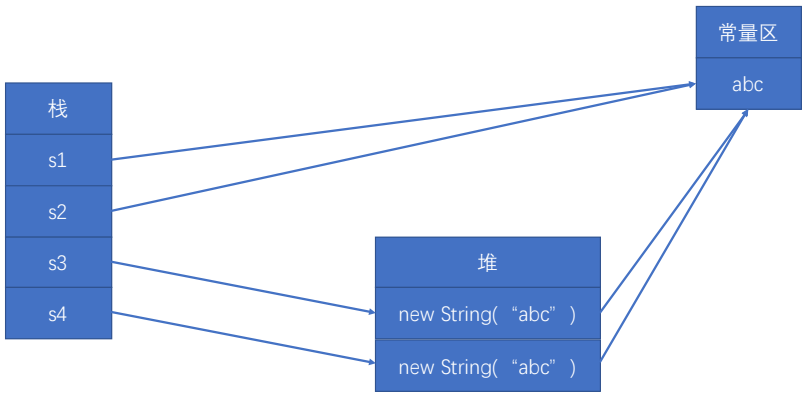
String s3 = new String(“abc”); //在堆中创建新的对象

String s4 = new String(“abc”); //在堆中又创建一个新的对象

为了便于理解，可以把 String s = new String(“abc”)语句的执行人为地分解成两个过程：第一个过程是新建对象的过程，即 new String(“abc”)；第二个过程是赋值的过程，即 String s =。由于第二个过程只是定义了一个名为 s 的 String 类型的变量，将一个 String 类型对象的引用赋值给 s，因此在这个过程中不会创建新的对象。第一个过程中 new String(“abc”)会调用 String 类的构造函数：

```
public String(String original) {
    //body
}
```

在调用这个构造函数时，传入了一个字符串常量，因此语句 new String(“abc”)也就等价于 “abc” 和 new String()两个操作了。若在字符串池中不存在 “abc”，则会创建一个字符串常量 “abc”，并将其添加到字符串池中；若存在，则不创建，然后 new String()会在堆中创建一个新的对象，所以 s3 与 s4 指向的是堆中不同的 String 对象，地址自然也不相同。



- Java 中数组是不是对象？

数组是指具有相同类型的数据的集合，它们一般具有固定的长度，并且在内存中占据连续的空间。在 C/C++语言中，数组只是一个指针，这个指针指向了数组的首元素，既没有属性也没有方法可以调用，而在 Java 语言中，数组不仅有其自己的属性(例如 length 属性)，也有一些方法可以被调用(例如 clone 方法)。由于对象的特点是封装了一些数据，同时提供了一些属性和方法，从这个角度来讲，数组是对象。**每个数组类型都有其对应的类型**，可以通过 instanceof 来判断数据的类型。

- finally 块中的代码什么时候被执行？

在 Java 语言的异常处理中，finally 块的作用就是为了保证无论出现什么情况，finally 块里的代码一定会被执行。由于程序执行 return 就意味着结束对当前函数的调用并跳出这个函数体，因此任何语句要执行都只能在 return 前执行（除非碰到 exit 函数），因此 **finally 块里的代码也是在 return 前执行的**。此外，如果 try-finally 或者 catch-finally 中都有 return，那么 **finally 块中的 return 将会覆盖别处的 return 语句，最终返回到调用者那里的是 finally 中 return 的值**。

由于在一个方法内部定义的变量都存储在栈中，当这个函数结束后，其对应的栈就会被回收，此时在其方法体中定义的变量将不存在了，因此 **return 在返回时不是直接返回变量的值，而是复制一份，然后返回。因此，对于基本类型的数据，在 finally 块中改变 return 的值对返回值没有任何影响，而对引用类型的数据会有影响**。

- 出现在 Java 程序中的 finally 块是不是一定会被执行？

  不一定。

  （1）当程序在进入 try 语句块之前就出现异常时，会直接结束，不会执行 finally 块中的代码。

  （2）当程序在 try 块中强制退出时也不会去执行 finally 块中的代码。

- 异常处理的原理是什么？

  异常是指程序运行时（非编译时）所发生的非正常情况或错误，当程序违反了语义规则时，JVM 就会讲出现的错误表示为一个异常并抛出。这个异常可以在 catch 程序块中进行捕获，然后进行处理。而异常处理的目的则是为了提高程序的安全性与鲁棒性。

  Java 语言把异常当作对象来处理，并定义了一个基类（java.lang.**Throwable**）作为所有异常的父类。在 Java API 中，已经定义了许多异常类，这些异常类分为 **Error**（错误）和 **Exception**（异常）两大类。

  违反语义规则包括两种情况：一种是 Java 类库内置的语义检查，例如当数组下标越界时，会引发 IndexOutOfBoundsException，当访问 null 的对象时，会引发 NullPointerException ;另一种情况是 Java 允许开发人员扩展这种语义检查，开发人员可以创建自己的异常类（所有异常都是 java.lang.Throwable 的子类），并自由选择在何时用 throw 关键字抛出异常。

- 运行时异常和普通异常有什么区别？

Java 提供了两种错误的异常类，分别为 Error 和 Exception，且它们拥有共同的父类——Throwable。

Error 表示程序运行期间出现了非常严重的错误，并且该错误是不可恢复的，由于这属于 **JVM 层次的严重错误**，因此这种错误是会**导致程序终止执行**的。此外，编译器不会检查 Error 是否被处理，因此在程序中不推荐去捕获 Error 类型的异常，主要原因是运行时异常是由于逻辑错误导致的，属于应该解决的错误，也就是说，一个正确的程序中是不应该存在 Error 的。**OutOfMemoryError**、**ThreadDeath** 等都属于错误。当这些异常发生时，JVM 一般会选择将线程终止。

Exception 表示可恢复的异常，是编译器可以捕捉到的。它包含两种类型：**检查异常**（checked exception）和**运行时异常**（runtime exception）。

（1）检查异常是在程序中最经常碰到的异常。所有继承自 Exception 并且不是运行时异常的异常都是检查异常，比如最常见的 IO 异常和 SQL 异常。这种异常都发生在编译阶段，Java 编译器强制程序去捕获此类型的异常，即把可能会出现这些异常的代码放到 try 块中，把对异常的处理的代码放到 catch 块中。这种异常一般在如下几种情况中使用：

①异常的发生并不会导致程序出错，进行处理后可以继续执行后续的操作，例如，当连接数据库失败后，可以重新连接后进行后续操作。

②程序依赖于不可靠的外部条件，例如系统 IO。

（2）运行时异常不同于检查异常，编译器没有强制对其进行捕获并处理。如果不对这种异常进行处理，当出现这种异常时，会由 JVM 来处理，例如 NullPointerException 异常，它就是运行时异常。在 Java 语言中，最常见的运行时异常包括 NullPointerException（空指针异常）、ClassCastException（类型转换异常）、ArrayIndexOutOfBoundsException（数组越界异常）、ArrayStoreException（数组存储异常）、BufferOverflowException（缓冲区溢出异常）、ArithmeticException（算术异常）等。

出现运行时异常后，系统会把异常一直往上层抛出，直到遇到处理代码为止。若没有处理块，则抛到最上层；如果是多线程就用 Thread.run()方法抛出，如果是单线程，就用 main()方法抛出。抛出之后，如果是线程，那么这个线程也就退出了。如果是主程序抛出的异常，那么整个程序也就退出了。所以，如果不对运行时的异常进行处理，后果是非常严重的，一旦发生，要么是线程中止，要么是程序终止。

在使用异常处理时，还需要注意以下几个问题：

①Java 异常处理用到了多态的概念，如果在异常处理过程中，先捕获了基类，

然后再捕获子类，那么捕获子类的代码块将永远不会被执行。因此，在进行异常捕获时，正确的写法是：先捕获子类，再捕获基类的异常信息。

②尽早抛出异常，同时对捕获的异常进行处理，或者从错误中恢复，或者让程序继续执行。

③可以根据实际的需求自定义异常类，这些自定义的异常类只要继承自 Exception 即可。

④异常能处理就处理，不能处理就抛出。

- Java IO 流的实现机制是什么？

在 Java 语言中，输入和输出都被称为抽象的流，流可以被看作一组有序的字节集合，即数据在两设备之间的传输。

流的本质是数据传输，根据处理数据类型的不同，流可以分为两大类：字节流和字符流。字节流以字节（8bit）为单位，包含两个抽象类：InputStream（输入流）和 OutputStream（输出流）。字符流以字符（16bit）为单位，根据码表映射字符，一次可以读多个字节，它包含两个抽象类：Reader（输入流）和 Writer（输出流）。字节流和字符流最主要的区别为：字节流在处理输入输出时不会用到缓存，而字符流用到了缓存。

- Java NIO 是什么？

NIO 通过 Selector、Channel 和 Buffer 来实现非阻塞的 IO 操作。

NIO 非阻塞的实现主要采用了 Reactor（反应器）设计模式，这个设计模式与 Observer（观察者）设计模式类似，只不过 Observer 设计模式只能处理一个事件源，而 Reactor 设计模式可以用来处理多个事件源。

Channel 可以被看作是一个双向的非阻塞的通道，在通道的两边都可以进行数据的读写操作。Selector 实现了用一个线程来管理多个通道，它类似于一个观察者。Buffer 用来保存数据，可以用来存放从 Channel 读取的数据，也可以存放使用 Channel 进行发送的数据。

- 什么是 Java 序列化？

Java 提供了两种对象持久化的方式，分别为序列化和外部序列化。

（1）序列化（Serialization）

在分布式环境下，当进行远程通信时，无论是何种类型的数据，都会以二进制序列的形式在网络上传送。**序列化是一种将对象以一连串的字节描述的过程，**

用于解决在对对象流进行读写操作时所引发的问题。序列化可以将对象的状态写在流里进行网络传输，或者保存到文件、数据库等系统里，并在需要时把该流读取出来重新构造一个相同的对象。

如何实现序列化呢？其实，所有要实现序列化的类都必须实现 Serializable 接口，Serializable 接口位于 java.lang 包中，它里面没有包含任何方法。使用一个输出流（例如 FileOutputStream）来构造一个 ObjectOutputStream（对象流）对象，紧接着，使用该对象的 writeObject(Object obj)方法就可以将 obj 对象写出（即保存其状态），要恢复时可以使用其对应的输入流。

序列化有以下两个特点：

①**如果一个类能被序列化，那么它的子类也能够被序列化**。

②**由于 static 代表类的成员、transient 代表对象的临时数据，因此被声明为这两种类型的数据成员是不能够被序列化的**。

由于序列化的使用会影响系统的性能，因此如果不是必须要使用序列化，应尽可能不要使用序列化。那么在什么情况下需要使用该序列化呢？

①**需要通过网络来发送对象**，或**对象的状态需要被持久化到数据库或文件中**。

②序列化能实现**深复制**，即**可以复制引用的对象**。

与序列化相对的是反序列化，它将流转换为对象。在序列化与反序列化的过程中，serialVersionUID 起着非常重要的作用，每个类都有一个特定的 serialVersionUID，在反序列化的过程中，通过 serialVersionUID 来判定类的兼容性。**如果待反序列化的对象与目标对象的 serialVersionUID 不同，那么在反序列化时就会抛出 InvalidClassException 异常**。作为一个好的编程习惯，最好在被序列化的类中显式地声明 serialVersionUID（该字段必须定义为 static final）。自定义 serialVersionUID 主要有如下 3 个优点：

①**提高程序的运行效率**。如果在类中未显式声明 serialVersionUID，那么在序列化时会通过计算得到一个 serialVersionUID 值。通过显式声明 serialVersionUID 的方式省去了计算的过程，因此提高了程序的运行效率。

②**提高程序不同平台上的兼容性**。由于各个平台的编译器在计算 serialVersionUID 时完全有可能会采用不同的计算方式，这就会导致在一个平台上序列化的对象在另外一个平台上将无法实现反序列化的操作。通过显式声明 serialVersionUID 的方法完全可以避免该问题的发生。

③**增强程序各个版本的可兼容性**。在默认情况下，每个类都有唯一的 serialVersionUID，因此，当后期对类进行修改时（例如加入新的属性），类的 serialVersionUID 值将会发生变化，这将会导致类在修改前对象序列化的文件在

修改后将无法进行反序列化操作。同样，通过显式声明 serialVersionUID 也会解决这个问题。

（2）外部序列化（Externalizable）

外部序列化与序列化主要的区别在于序列化是内置的 API，只需要实现 Serializable 接口，开发人员不需要编写任何代码就可以实现对象的序列化，而使用外部序列化时，Externalizable 接口中的读写方法必须由开发人员来实现。因此与实现 Serializable 接口的方法相比，使用 Externalizable 编写程序的难度更大，但是由于把控制权交给了开发人员，在编程时由更多的灵活性，对需要持久化的那些属性可以进行控制，可能会提高性能。

- 在用接口 Serializable 实现序列化时，这个类中的所有属性都会被序列化，那么怎样才能实现只序列化部分属性呢？

一种方法为**实现 Externalizable 接口**，开发人员可以根据实际需求来实现 readExternal 与 writeExternal 方法来控制序列化与反序列化所使用的属性，这种方法的缺点为增加了编程的难度。另一种方法为**使用关键字 transient 来控制序列化的属性**。被 transient 修饰的属性是临时的，不会被序列化。因此，可以把不需要被序列化的属性用 transient 来修饰。

当一个对象不再被引用后就成为垃圾可以被回收，但是**线程就算没有被引用也可以独立运行**。

- Java 引起内存泄漏的原因有哪些？

（1）**静态集合类**，例如 HashMap 和 Vector。如果这些容器为静态的，由于它们的声明周期与程序一致，那么容器中的对象在程序结束之前将不能被释放，从而造成内存泄漏。

（2）各种连接，例如**数据库连接**、**网络连接**以及 **IO 连接**等。在对数据库进行操作的过程中，首先需要建立与数据库的连接，当不再使用时，需要调用 close 方法来释放与数据库的连接。只有连接被关闭后，垃圾回收器才会回收对应的对象。否则，如果在访问数据库的过程中，对 Connection、Statement 或 ResultSet 不显式地关闭，将会造成大量的对象无法被回收，从而引起内存泄漏。

（3）**监听器**。在 Java 语言中，往往会使用到监听器。通常一个应用中会用到多个监听器，但在释放对象的同时往往没有相应地删除监听器，这也可能导致内存泄漏。

（4）**变量不合理的作用域**。如果一个变量定义的作用范围大于其使用的范围，很有可能会造成内存泄漏。如果没有及时地把对象设置为 null，很有可能会导致内存泄漏。

（5）**单例模式**可能会造成内存泄漏。

- 什么是迭代器？

迭代器（Iterator）是一个对象，它的工作是遍历并选择序列中的对象，它提供了一种访问一个容器对象中的各个元素，而又不必暴露该对象内部细节的方法。通过迭代器，开发人员不需要了解容器底层的结构，就可以实现对容器的遍历。由于创建迭代器的代价小，因此迭代器通常被称为**轻量级的容器**。

在使用 iterator()方法时经常会遇到 ConcurrentModificationException 一次，这通常是由于在使用 Iterator 遍历容器的同时又对容器做增加或删除操作所导致的，或者由于多线程操作导致，当一个线程使用迭代器遍历容器的同时，另外一个线程对这个容器进行增加或删除操作。

当调用容器的 iterator()方法返回 Iterator 对象时，把容器中包含对象的个数赋值给了一个变量 expectedModCount，在调用 next()方法时会比较变量 expectedModCount 与容器中实际对象的个数 modCount 的值是否相等，若二者不相等，则会抛出 ConcurrentModificationException 异常，因此在使用 Iterator 遍历容器的过程中，如果对容器进行增加或删除操作，就会改变容器中对象的数量，从而导致抛出异常。解决方法如下：在遍历的过程中把需要删除的对象保存到一个集合中，等遍历结束后再调用 removeAll()方法来删除，或者使用 iter.remove()方法。

以上主要介绍了单线程的解决方案，那么多线程访问容器的过程中抛出 ConcurrentModificationException 异常又该指明解决呢？

①在 JDK 1.5 版本引入了线程安全的容器，比如 **ConcurrentHashMap** 和 **CopyOnWriteArrayList** 等。可以使用这些线程安全的容器来代替非线程安全的容器。

②把在使用迭代器遍历容器时对容器的操作放到 **synchronized 代码块**中，但是当引用程序并发程度比较高时，这会严重影响程序的性能。

- Callable 接口与 Runnable 接口有什么区别？

Callable 接口实际是属于 Executor 框架中的功能类，Callable 接口与 Runnable 接口的功能类似，但提供了比 Runnable 更强大的功能，主要表现为以

下 3 点：

①Callable 可以在任务结束后提供一个**返回值**，Runnable 无法提供这个功能。

②Callable 中的 call()方法**可以抛出异常**，而 Runnable 的 run()方法不能抛出异常。

③运行 Callable 可以拿到一个 **Future 对象**，Future 对象表示异步运算的结果，它提供了检查计算是否完成的方法。由于线程属于异步计算模型，因此无法从别的线程中得到函数的返回值，在这种情况下，就可以使用 Future 来监视目标线程调用 call()方法的情况，**当调用 Future 的 get()方法以获取结果时，当前线程就会阻塞，直到 call()方法结束返回结果**。

一个类可以同时继承 Thread 与实现 Runnable 接口。

- sleep()方法与 wait()方法有什么区别？

sleep()是使线程暂停执行一段时间的方法。wait()也是一种使线程暂停执行的方法，例如，当线程交互时，如果线程对一个同步对象 x 发出一个 wait()调用请求，那么该线程会暂停执行，被调对象进入等待状态，直到被唤醒或等待时间超时。

具体而言，sleep()方法与 wait()方法的区别主要表现在以下几个方面：

①原理不同。sleep()方法是 Thread 类的静态方法，是线程用来控制自身流程的，它会使此线程暂停执行一段时间，而把执行计划让给其他线程，等到计时时间一到，此线程会自动“苏醒”。wait()方法是 Object 类的方法，用于线程间的通信，这个方法会使当前拥有该对象锁的进程等待，直到其他线程调用 notify()方法（或 notifyAll()方法）时才“醒”来，不过开发人员也可以给它指定一个时间，自动“醒”来。与 wait()方法配套的方法还有 notify()方法和 notifyAll()。

②对锁的处理机制不同。由于 sleep()方法的主要作用是让线程暂停执行一段时间，时间一到则自动恢复，不涉及线程间的通信，因此，调用 sleep()方法并不会释放锁。而 wait()方法则不同，当调用 wait()方法后，线程会释放掉它所占用的锁，从而使线程所在对象中的其他 synchronized 数据可被别的线程使用。

③使用区域不同。由于 wait()方法的特殊意义，因此它必须放在同步控制方法或者同步语句块中使用，而 sleep()方法则可以放在任何地方使用。

sleep()方法必须捕获异常，而 wait()、notify()以及 notifyall()不需要捕获异常。在 sleep 的过程中，有可能被其他对象调用它的 interrupt()，产生 InterruptedException 异常。

在 sleep 的过程中，有可能被其他对象调用它的 interrupt()，产生 InterruptedException 异常。

由于 sleep 不会释放“锁标志”，容易导致死锁问题的发生，因此，一般情况下，不推荐使用 sleep()方法，而推荐使用 wait()方法。

- sleep()方法与 yield()方法有什么区别？

①sleep()方法给其他线程运行机会时不考虑线程的优先级，因此会给低优先级的线程以运行的机会，而 yield()方法只会给相同优先级或更高优先级的线程以运行的机会。

②线程执行 sleep()方法后会转入阻塞状态，所以，执行 sleep()方法的线程在指定的时间内肯定不会被执行，而 yield()方法只是使当前线程重新回到可执行状态，所以执行 yield()方法的线程有可能在进入到可执行状态后马上又被执行。

③sleep()方法声明抛出 InterruptedException，而 yield()方法没有声明任何一次。

④sleep()方法比 yield()方法（跟操作系统有关）具有更好的移植性。

- 终止线程的方法有哪些？

（1）**设置一个 volatile 类型的 flag 标志**来控制循环是否执行，通过这种方法来让线程离开 run()方法从而终止线程。

（2）当线程处于非运行状态时（当 sleep()方法被调用或当 wait()方法被调用或当被 I/O 阻塞时），设置 volatile 类型 flag 标志位的方法就不可用了，此时可以**使用 interrupt()方法来打破阻塞的情况**，当 interrupt()方法被调用时，会抛出 InterruptedException 异常，可以通过在 run()方法中捕获这个异常来让线程安全退出。

（3）如果程序因为 I/O 而停滞，进入非运行状态，基本上要等到 I/O 完成才能离开这个状态，在这种情况下，无法使用 interrupt()方法来使程序离开 run()方法。这就需要使用一个替代的方法，基本思路也是**触发一个异常，而这个异常与所使用的 I/O 相关**，例如，如果使用 readLine()方法在等待网络上的一个信息，此时线程处于阻塞状态。让程序离开 run()方法就是**使用 close()方法来关闭流，在这种情况下会引发 IOException 异常**，run()方法可以通过捕获这个异常来安全地结束线程。

- 如何通过 JDBC 访问数据库？

①加载 JDBC 驱动器。将数据库的 JDBC 驱动加载到 classpath 中，在基于 JavaEE 的 Web 应用开发过程中，通常要把目标数据库产品的 JDBC 驱动复制到 WEB-INF/lib 下。

②加载 JDBC 驱动，并将其注册到 DriverManager 中。一般使用反射 Class.forName(String driverName)。

③建立数据库连接，取得 Connection 对象。一般通过 DriverManager.getConnection(url ,username, passwd)方法实现。

④建立 Statement 对象或是 PrepareStatement 对象。

⑤执行 SQL 语句。

⑥访问结果集 ResultSet 对象。

⑦依次将 ResultSet、Statement、PreparedStatement、Connection 对象关闭，释放掉所占用资源。原因在于 JDBC 驱动在底层通常都是通过网络 IO 实现 SQL 命令与数据传输的。

脏读：一个事务读取了另一个事务尚未提交的数据。例如，当事务 A 与事务 B 并发执行时，当事务 A 更新后，事务 B 查询读取到 A 尚未提交的数据，此时事务 A 回滚，则事务 B 读到的数据是无效的“脏”数据。

不可重复读：一个事务的操作导致另一个事务前后两次读取到不同的数据。例如，当事务 A 与事务 B 并发执行时，当事务 B 查询读取数据后，事务 A 更新操作更改事务 B 查询到的数据，此时事务 B 再次读取该数据，发现前后两次的数据不一样。

幻读：一个事务的操作导致另一个事务前后两次查询的结果数据量不同。例如，当事务 A 与事务 B 并发执行时，当事务 B 查询读取数据后，事务 A 新增或删除了一条满足事务 A 的查询条件的记录，此时，事务 B 再次查询，发现查询到前次不存在的记录，或者前次的某个记录不见了。

- Class.forName 的作用是什么？

在 Java 语言中，任何类只有被装载到 JVM 上才能运行。Class.forName()方法的作用就是把类加载到 JVM 中，它会**返回一个带有给定字符串名的类或接口相关联的 Class 对象**，并且 **JVM 会加载这个类**，同时 **JVM 会执行该类的静态代码段**。

- Statement、PreparedStatement 和 CallableStatement 有什么区别？

Statement 用于执行不带参数的简单 SQL 语句，并返回它所生成结果的对象，每次执行 SQL 语句时，数据库都要编译该 SQL 语句。

PreparedStatement 表示预编译的 SQL 语句的对象，用于执行带参数的预编译 SQL 语句。

CallableStatement 提供了用来调用数据库中存储过程的接口，如果有输出参数要注册，说明是输出参数。

虽然 Statement 对象与 PreparedStatement 对象能够完成相同的功能，但相比之下，PreparedStatement 具有以下优点：

①效率更高。在使用 PreparedStatement 对象执行 SQL 命令时，命令会被数据库进行编译和解析，并放到命令缓冲区。然后，每当执行同一个 PreparedStatement 对象时，由于在缓冲区中可以发现预编译的命令，虽然它会被再解析一次，但不会被再次编译，是可以重复使用的，能够有效提高系统性能，因此，如果要执行插入、更新、删除等操作，最好使用 PreparedStatement。

②代码可读性和可维护性更好。

③安全性更好。使用 PreparedStatement 能够预防 SQL 注入攻击，所谓 SQL 注入，指的是通过把 SQL 命令插入到 Web 表单递交或输入域名或页面请求的查询字符串，最终达到欺骗服务器，达到执行恶意 SQL 命令的目的。注入只对 SQL 语句的编译过程有破坏作用，而执行阶段只是把输入串作为数据处理，不再需要对 SQL 语句进行解析，因此也就避免了类似 select * from user where name = 'aa' and password = 'bb' or 1 = 1 的 SQL 注入问题的发生。

CallableStatement 由 prepareCall()方法所创建，它为所有 DBMS（Database Management System）提供了一种以标准形式调用已储存过程的方法。它从 PreparedStatement 中继承了用于处理输入参数的方法，而且还增加了调用数据库中的存储过程和函数以及设置输出类型参数的功能。

- getString()方法与 getObject()方法有什么区别？

JDBC 提供了 getString()、getInt()和 getData()等方法从 ResultSet 中获取数据，当查询结果集中的数据量较小时，不用考虑性能，使用这些方法完全能够满足需求，但是当查询结果集中的数据量非常大时，则会抛出异常。而通常情况下，使用 getObject()方法就可以解决这个问题。

getString()或 getInt()等方法在被调用时，程序会一次性地把数据都放到内存中，然后通过调用 ResultSet 的 next()和 getString()等方法来获取数据。当数据量大到内存中放不下时就会抛出异常，而**使用 getObject()方法就不会有这种问**

题，因为数据不会一次性被读到内存中，每次调用时会直接从数据库中去获取数据，因此使用这种方法不会因为数据量过大而出错。

- 页面请求的工作流程是怎样的？

（1）用户通过浏览器输入链接地址来请求所需的资源。

（2）浏览器接受用户的请求，并把该请求组装成指定的格式发送给服务器端，客户端与服务器端之间通过 HTTP 来完成具体的交互。其中请求的数据流中主要包含 HTTP 请求方法、请求的网址以及请求的一些参数信息。

（3）服务器接收到客户端发来的请求，并查找用户所需的资源。

（4）服务器查找到用户请求的资源后，把该资源返回给客户端。

（5）服务器通过把响应消息组装成特定的消息格式后返回给客户端，这个过程通过 HTTP 来完成。响应的数据流主要包含状态编码，Content-type，响应消息的内容。

（6）浏览器对 HTML 进行解析，并把响应结果展现给用户。

- 什么是 Servlet？

Servlet 是采**用 Java 语言编写的服务器端程序**，它**运行于 Web 服务器中的 Servlet 容器中**，其主要功能是**提供请求/响应的 Web 服务器模式**，可以**生成动态的 Web 内容**，而这正是 HTML 所不具备的内容。

- Servlet 处理客户端请求的步骤？

（1）用户通过单击一个链接来向 Servlet 发起请求。

（2）Web 服务器接收到该请求后，会把该请求交给相应的容器来处理，当容器发现这是对 Servlet 发起的请求后，容器此时会**创建两个对象：HttpServletResponse 和 HttpServletRequest**。

（3）容器可以根据请求消息中的 URL 消息找到对应的 Servlet，然后针对该请求**创建一个单独的线程**，同时把第（2）步中创建的两个对象以参数的形式传递到新创建的线程中。

（4）容器调用 Servlet 的 service()方法来完成对用户请求的响应，service()方法会调用 doPost()或 doGet()方法来完成具体的响应任务，同时把生成的动态页面返回给容器。

（5）容器把响应消息组装成 HTTP 格式返回给客户端。此时，这个线程运行结束，同时删除第（2）步创建的两个对象。

- 什么是 Servlet 的生命周期？

（1）**加载**。容器通过类加载器使用 Servlet 类对应的文件来加载 Servlet。

（2）**创建**。通过调用 Servlet 的构造函数来创建一个 Servlet 实例。

（3）**初始化**。通过调用 Servlet 的 init()方法来完成初始化工作，这个方法是在 Servlet 已被创建但向客户端提供服务之前调用的，需要注意的是，**init()方法只会被调用一次**。

（4）**处理客户请求**。Servlet 一旦被创建后，它就可以为客户端提供服务了。每当有新的客户请求到来时，容器都会创建一个新的线程来处理该请求，接着会调用 Servlet 的 service()方法来完成客户端的请求，当然，service()方法会根据请求的 method 属性值的不同调用决定是调用 doGet()方法还是调用 doPost()方法来完成具体的响应。

（5）**卸载**。容器在卸载 Servlet 之前需要调用 destroy()方法，让 Servlet 自己释放其占用的系统资源，一旦 destroy 方法被调用，容器就不会再向这个 Servlet 发送任何请求消息了。如果容器需要这个 Servlet，那么就必须重新创建并初始化一个实例。需要注意的是，**destroy()方法只会被调用一次**。

- JSP 有哪些优点？

JSP 是由 Sun 公司倡导、许多企业参与建立起来的一种动态技术标准，从本质上讲，就是嵌入了 Java 代码的 HTML 文件。在引入 JSP 之前，在 Web 应用程序中，所有业务逻辑和 HTML 的响应都是在 Servlet 中实现的，但在使用这种方式时，存在一个比较大的缺陷：**必须把给用户响应的视图组装成一个很长的 HTML 格式的字符串写入 println()方法中**。所以，使用这种方式编程，在编写一段很长的 HTML 字符串时非常容易出错，而且代码的可读性也非常差，同时由于业务逻辑与视图没有分离，系统的可扩展性、可维护性都较差。

JSP 的引入在一定程序上解决了 Servlet 存在的缺点。其实现理念是让每个 Servlet 只负责其对应的业务逻辑的处理，让 JSP 来负责用户的 HTML 显示，因此**实现了业务逻辑与视图实现的分离**，从而极大地提高了系统的可扩展性。

- JSP 与 Servlet 有何异同？

JSP 与 Servlet 的相同点为：JSP 可以被看作一个特殊的 Servlet，它只不过是对 Servlet 的扩展，只要是 JSP 可以完成的工作，使用 Servlet 都可以完成。由于 JSP 页面最终要被转换成 Servlet 来运行，所以处理请求实际上是编译后的 Servlet。

JSP 与 Servlet 的不同点为：(1) Servlet 的实现方式是在 Java 中嵌入 HTML 代码，编写和修改 HTML 非常不方便，所以它比较适合做流程控制、业务处理；而 JSP 的实现方式为在 HTML 中嵌入 Java 代码，比较适合页面的显示。(2) Servlet 中没有内置对象，JSP 中的内置对象都是必须通过 HttpServletRequest 对象、HttpServletResponse 对象以及 HttpServlet 对象得到。

- MVC 的处理过程？

对于每一个用户输入的请求，先被控制器接收，并决定由哪个模型来进行处理，然后模型通过业务逻辑层处理用户的请求并返回数据，最后控制器用相应的视图格式化模型返回的数据，并通过显示页面呈现给客户。

- MVC 设计结构的优点？

(1) **低耦合性**。由于视图层和业务层分离，这样就使得修改视图层代码时不需要重新编译模型和控制器的代码，同样，一个应用业务规则的改变只需要改动 MVC 的模型层即可。因为模型与控制器和视图相分离，所以很容易改变应用程序的数据层和业务规则。

(2) **高重用性和可适用性**。由于技术的不断进步，现在访问应用程序可以有越来越多的方式。MVC 模式允许适用各种不同样式的视图来访问同一个服务器端的代码。它包括任何 Web 浏览器或者无线浏览器。由于模型返回的数据没有进行格式化，因此同样的构件能被不同的界面适用。

(3) **较低的生命周期成本**。MVC 使得开发和维护用户接口的技术难度降低。

(4) **部署快速**。适用 MVC 模式可以大大缩减开发时间，这使得后台开发人员集中精力于业务逻辑上，使界面开发人员集中精力于表现形式上。

(5) **可维护性**。分离视图层和业务逻辑层也使得 Web 应用更易于维护和修改。

(6) **有利于软件工程化管理**。由于采用了分层思想，每一层不同的应用具有某些相同的特征，有利于通过工程化、工具化管理程序代码。

- Servlet 中 forward 和 redirect 有什么区别？

forward 是**服务器内部的重定向**，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，而客户端并不知道，因此在客户端浏览器的地址栏中不会显示转向后的地址，还是原来的地址。由于在整个定向的过程中用的是同一个 Request，因此 forward 会将 Request 的信息带到被定向的 JSP 或 Servlet 中使

用。

redirect 则是**客户端的重定向**，是完全的跳转，即客户端浏览器会获取到跳转后的地址，然后重新发送请求，因此浏览器中会显示跳转后的地址。同时，由于这种方式比 forward 方式多了一次网络请求，因此其效率要低于 forward 方式。需要注意的是，客户端的重定向可以通过设置特定的 HTTP 头或写 JavaScript 脚本实现。

- JSP 的内置对象有哪些？

名称	描述
request	客户端请求，此请求包含来自 GET/POST 请求的参数。客户端的请求信息被封装在 request 对象中，通过它才能了解到客户的需求，然后做出响应，因此 request 对象是用来获取请求参数的非常重要的途径。它是 HttpServletRequest 类的实例。
response	用来表示服务器端对客户端的响应，将 Web 服务器处理后的结果返回给客户端。但在 JSP 中很少直接使用到它。它是 HttpServletResponse 类的实例。
pageContext	提供了对 JSP 页面的所有对象及命名空间的访问，也就是说，用它可以访问到本页面中的所有其他对象，例如前面已经描述的 request、response 以及后面要介绍的 session 和 application 对象等。它的本类名也叫 PageContext。
session	用来表示客户端与服务器的一次会话。从客户端与 Web 服务器建立连接的时候会话开始，直到关闭浏览器时结束会话。它是 HttpSession 类的实例。
application	代表 JSP 所属的 Web 应用本身。application 对象可存放全局变量，因此可以实现用户间的数据共享。它的生命周期与服务器的生命周期一致，也就是说，服务器启动后这个对象即被创建出来，直到服务器停止后这个对象的生命周期才结束。在任何地方，对此对象属性的操作都将影响到其他用户对此的访问。它是 ServletContext 类的实例。
out	用于在客户端浏览器内输出信息。它是 JspWriter 类的实例。
config	主要作用是取得服务器的配置信息。当一个 Servlet 初始化时，容器把某些信息通过 config 对象传递给这个 Servlet，Servlet 可以使用这个对象获取所需的配置信息。
page	表示当前 JSP 页面，类似于 Java 中的 this 指针。它是 java.lang.Object 类的实例。
exception	用来表示异常。当一个页面在运行过程中发生了异常，就会产生这个对象。如果 JSP 需要使用这个对象，就必须把 isErrorPage 设为 true，否则将无法编译。它是 java.lang.Throwable 的对象。

- 什么是 Ajax？

Ajax 是一个结合了 Java 技术、XML 以及 JavaScript 的编程技术，其主要目的是在不刷新页面的情况下通过与服务器进行少量数据的交互来提高页面的交互性，减少响应时间，从而改善用户体验。使用 Ajax 技术后，页面就不需要在每次用户提交修改时重新加载了。

由于 Ajax 可以只向服务器发送并取回必需的数据内容，使得数据交互量大幅降低，从而降低了服务器的网络负载；其次，由于它通过使用 SOAP 或其他一些基于 XML 的 Web Service 接口，在客户端采用 JavaScript 处理来自服务器的响应，也降低了 Web 服务器的处理时间；最后，由于不需要重新加载整个页面，因此系统有更短的响应时间，而这有利于提高系统的稳定性与可用性，从而增强用户的满意度。

Ajax 是客户端技术，其核心是 JavaScript 对象 XmlHttpRequest，该对象是一种支持异步请求的技术，它使得开发人员可以使用 JavaScript 向服务器提出请求并处理响应，而不阻塞用户。

- cookie 和 session 有什么区别？

cookie 是在 HTTP 下，服务器或脚本可以维护客户工作站上信息的一种方式。它是由 Web 服务器保存在用户浏览器上的小文件，可以包含有关用户的信息。session 是指用来在客户端与服务器端之间保持状态的解决方案以及存储结构。

（1）cookie 机制采用的是在**客户端保持状态**的方案，即数据存放在客户的浏览器上；而 session 机制采用的是在**服务器端保持状态**的方案，即数据放在服务器上。

（2）**cookie 安全性不够**。由于 cookie 信息存放在客户端，其他人可以很容易地得到存放在本地的 cookie，并进行 cookie 欺骗；而 session 信息存放在服务器端，因此较为安全。

（3）**cookie 性能更高一些**。由于 session 会在一定时间内保存在服务器上，因此当访问量增多时，会降低服务器的性能。

（4）单个 cookie 保存的数据不能超过 4KB，很多浏览器都限制一个站点最多保存 20 个 cookie，而 session 不存在此问题。

一般情况下，将用户登录信息等重要信息存放至 session 中，而其他需要保留的信息可以放在 cookie 中。

Web 服务器是指驻留在 Internet 上的计算机程序。它是一种服务程序，其主要工作是接收来自于客户端的请求，然后把对请求的处理结果返回给客户端。用户可以通过浏览器来请求所需资源，这些资源可以是 HTML 页面、图片、音频、视频或者 PDF 文档等，Web 服务器接收到请求后会去查找用户请求的资源，然后把找到的资源返回给用户。需要注意的是，它是一个被动的程序，只有当收到来自客户端的请求后，才会发送响应。目前最流行的两大 Web 服务器是：IIS（Internet Information Services）和 Apache Http Server。

Web 容器也被叫作 **Web 应用服务器**，它是一种服务程序，用来给运行在其中的程序（例如 Servlet、JSP 等）提供一个运行环境。**由于 Web 容器的存在，Servlet 只需要关注业务逻辑的处理而不用关注与客户端的交互，因为这些交互都已经由 Web 容器来协助完成了**。常见的 Web 容器有 Tomcat、JBoss、WebLogic 和 WebSphere 等。

当 Web 服务器（例如 Apache）接收到一个客户端对 Servlet 的请求后，不是直接把这些请求交给 Servlet 来处理，而是交给 Web 容器来处理，由 Web 容器负责给 Servlet 提供 HTTP 请求与响应对象，同时容器调用 Servlet 的 doGet()和 doPost()方法来处理用户的请求。

- delete 与 truncate 命令有哪些区别？

（1）truncate 是一个数据定义语言（DDL），它会被隐式地提交，一旦执行后将不能回滚。delete 执行的过程是每次从表中删除一行数据，同时将删除的操作以日志的形式进行保存，以便将来进行回滚操作。

（2）用 delete 操作后，被删除的数据占用的存储空间还在，还**可以恢复**。而用 truncate 操作删除数据后，被删除的数据会立即释放占用的存储空间，被删除的数据是**不能被恢复**的。

（3）truncate 的执行速度比 delete 快。

- 内连接与外连接有什么区别？

内连接，也被称为自然连接，**只有两个表相匹配的行才能在结果集中出现**。返回的结果集选取了两个表中所有相匹配的数据，舍弃了不匹配的数据。由于内连接是从结果表中删除与其他连接表中没有匹配的所有行，所以内连接可能会造成信息的丢失。

内连接是保证两个表中的所有行都满足连接条件，而外连接则不然。**外连接**

不仅包含符合连接条件的行，而且还包括左表（左外连接时）、右表（右外连接时）或两个边接表（全外连接）中的所有数据行。SQL 的外连接共有 3 种类型：左外连接(关键字为 LEFT OUTER JOIN)、右外连接(关键字为 RIGHT OUTER JOIN)和全外连接（关键字为 FULL OUTER JOIN)。

- 什么是存储过程？它与函数有什么区别与联系？

SQL 语句在执行时要先编译，然后再被执行。在大型数据库系统中，为了提高效率，将为了完成特定功能的 SQL 语句集进行编译优化后，存储在数据库服务器中，用户通过指定存储过程的名字来调用执行。

使用存储过程可以增强 SQL 语言的功能和灵活性，由于可以用流程控制语句编写存储过程，有很强的灵活性，因此可以完成复杂的判断和运算，并且可以保证数据的安全性和完整性。同时，存储过程可以使没有权限的用户在控制之下间接地存取数据库，也保证了数据的安全。

要注意的是，存储过程不等于函数，二者虽然本质上没有区别，但还是有如下几个方面的不同：

（1）存储过程一般作为一个独立的部分来执行，而函数可以作为查询语句的一个部分来调用。由于函数可以返回一个对象，因此它可以在查询语句中位于 From 关键字的后面。

（2）一般而言，存储过程实现的功能较复杂，而函数实现的功能针对性较强。

（3）函数需要用括号包住输入的参数，且只能返回一个值或表对象，而存储过程可以返回多个参数。

(4) 函数可以嵌入在 SQL 中使用，可以在 select 中调用，存储过程则不行。

（5）函数不能直接操作实体表，只能操作内建表。

（6）存储过程在创建时即在服务器上进行了编译，其执行速度比函数快。

- 各种范式有什么区别？

按照“数据库规范化”对表进行设计，其目的就是减少数据库中的数据冗余。

（1）第一范式：列不可分。

（2）第二范式：主键完全依赖。

（3）第三范式：不存在依赖传递。

（4）第四范式：表中只有一个多值事实。

- 什么是触发器？

触发器是一种特殊类型的存储过程，它由事件触发，而不是程序调用或手工启动，当数据库有特殊的操作时，这些操作由数据库中的事件来触发，自动完成这些 SQL 语句。使用触发器可以用来保证数据的有效性和完整性，完成比约束更复杂的数据约束。

根据 SQL 语句的不同，触发器可分为 DML 触发器和 DLL 触发器。

DML 触发器是当数据库服务器发生数据操作语言事件时执行的存储过程，有 After 和 Instead Of 这两种触发器。After 触发器被激活触发是在记录改变之后进行的一种触发器。Instead Of 触发器是在记录变更之前，去执行触发器本身所定义的操作，而不是执行原来 SQL 语句里的操作。DLL 触发器是在响应数据定义语言事件时执行的存储过程。

触发器的主要作用表现为以下几个方面：

（1）增加**安全性**。

（2）利用触发器记录所进行的修改以及相关信息，**跟踪用户对数据库的操作**，实现审计。

（3）维护那些通过创建表时的声明约束不可能实现的复杂的完整性约束以及对数据库中特定事件进行监控与响应。

（4）实现复杂的非标准的数据库相关完整性规则、同步实时地复制表中的数据。

（5）触发器是自动的，它们在对表的数据做了任何修改之后就会被激活。

- 触发器分为事前触发和事后触发，这两者有什么区别？语句级触发和行级触发有什么区别？

事前触发发生在事件发生之前，用于验证一些条件或进行一些**准备工作**；事后触发发生在事件发生之后，做**收尾工作**。事前触发可以获得之前和新的字段值，而事后触发可以**保证事务的完整性**。语句级触发可以在**语句执行之前或之后**执行，而行级触发在**触发器所影响的每一行**触发一次。

- 什么是游标？

数据库中，游标提供了一种对从表中检索出的数据进行操作的灵活手段，它实际上是一种能从包含多条数据记录的结果集中每次提取一条记录的机制。

游标总是与一条 SQL 选择语句相关联，因为游标是由**结果集**和**结果集中指向特定记录的游标位置**组成的。当决定对结果集进行处理时，必须声明一个指向

该结果集的游标。

游标**允许应用程序对查询语句 select 返回的行结果集中的每一行进行相同或不同的操作**，而不是一次对整个结果集进行同一种操作；它还提供对**基于游标位置而对表中数据进行删除或更新的功能**；游标还**把作为面向集合的数据库管理系统和面向行的程序设计连接了起来**，使两种数据处理方式能够进行“沟通”。

- 如果数据库日志满了，会出现什么情况？

日志文件记录所有对数据库数据的修改，主要是保护数据库以防故障发生，以及恢复数据时使用。通过日志文件来记录数据库事务可以最大限度地保证数据的一致性和安全性，但一旦数据库中日志满了，就**只能执行查询等读操作，不能执行更改、备份等操作**，原因是任何写操作都要记录日志，也就是说，基本上处于不能用的状态。

- union 和 union all 有什么区别？

union 在进行表求并集后会**去掉重复的元素**，所以会对所产生的结果集进行**排序**运算，删除重复的记录再返回结果。

union all 则只是**简单地将两个结果集合并**后就返回结果，**不排序**。

- 什么是视图？

视图是由从数据库的基本表中选取出来的数据组成的逻辑窗口，与基本表不同，它是一个虚表。在数据库中，存放的只是视图的定义，而不存放视图包含的数据项，这些项目仍然存放在原来的基本表结构中。

视图的作用主要有以下几点：首先，可以**简化数据查询**语句；其次，可以使用户能**从多角度看待同一数据**；然后，通过引入视图可以**提高数据的安全性**；最后，视图提供了**一定程度的逻辑独立性**。

通过引入视图机制，用户可以将注意力集中在其关心的数据上（而非全部数据），这样就大大提高了用户效率与用户满意度，而且如果这些数据来源于多个基本表结构，或者数据不仅来自于基本表结构，还有一部分数据来源于其他视图，而且搜索条件又比较复杂时，需要编写的查询语句会比较繁琐，此时定义视图就可以使数据的查询语句变得简单可行。**定义视图可以将表与表之间的复杂的操作连接和搜索条件对用户不可见，用户只需要简单地对一个视图进行查询即可，故增加了数据的安全性，但不能提高查询的效率**。