1、面向对象的特征有哪些方面？
答：面向对象的特征主要有以下几个方面：
- 抽象：抽象是将一类对象的共同特征总结出来构造类的过程，包括数据抽象和行为抽象两方面。抽象只关注对象有哪些属性和行为，并不关注这些行为的细节是什么。
- 继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段（如果不能理解请阅读阎宏博士的《Java 与模式》或《设计模式精解》中关于桥梁模式的部分）。
- 封装：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口（可以想想普通洗衣机和全自动洗衣机的差别，明显全自动洗衣机封装更好因此操作起来更简单；我们现在使用的智能手机也是封装得足够好的，因为几个按键就搞定了所有的事情）。
- 多态性：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当 A 系统访问 B 系统提供的服务时，B 系统有多种提供服务的方式，但一切对 A 系统来说都是透明的（就像电动剃须刀是 A 系统，它的供电系统是 B 系统，B 系统可以使用电池供电或者用交流电，甚至还有可能是太阳能，A 系统只会通过 B 类对象调用供电的方法，但并不知道供电系统的底层实现是什么，究竟通过何种方式获得了动力）。方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西，要实现多态需要做两件事：1). 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；2). 对象造型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

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

修饰符	当前类	同包	子类	其他包
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

类的成员不写访问修饰时默认为 default。默认对于同一个包中的其他类相当于公开（public），对于不是同一个包中的其他类相当于私有（private）。受保护（protected）对子类相当于公开，对不是同一包中的没有父子关系的类相当于私有。Java 中，外部类的修饰符只能是 public 或默认，类的成员（包括内部类）的修饰符可以是以上四种。
3、String 是最基本的数据类型吗？
答：不是。Java 中的基本数据类型只有 8 个：byte、short、int、long、float、double、char、boolean；除了基本类型（primitive type）和枚举类型（enumeration type），剩下的都是引用类型（reference type）。

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

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

答：对于 short s1 = 1; s1 = s1 + 1;由于 1 是 int 类型，因此 s1+1 运算结果也是 int 型，需要强制转换类型才能赋值给 short 型。而 short s1 = 1; s1 += 1;可以正确编译，因为 s1+= 1;相当于 s1 = (short)(s1 + 1);其中有隐含的强制类型转换。

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

7、int 和 Integer 有什么区别？
答：Java 是一个近乎纯洁的面向对象编程语言，但是为了编程的方便还是引入了基本数据类型，但是为了能够将这些基本数据类型当成对象操作，Java 为每一个基本数据类型都引入了对应的包装类型（wrapper class），int 的包装类就是 Integer，从 Java 5 开始引入了自动装箱/拆箱机制，使得二者可以相互转换。
Java 为每个原始类型提供了包装类型：
- 原始类型: boolean，char，byte，short，int，long，float，double
- 包装类型：Boolean，Character，Byte，Short，Integer，Long，Float，Double

```
<span style="color:#333333;">class AutoUnboxingTest {

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

最近还遇到一个面试题，也是和自动装箱和拆箱有点关系的，代码如下所示：

```
public class Test03 {

    public static void main(String[] args) {
        Integer f1 = 100, f2 = 100, f3 = 150, f4 = 150;

        System.out.println(f1 == f2);
        System.out.println(f3 == f4);
    }
}
```

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

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

IntegerCache 是 Integer 的内部类，其代码如下所示：

```
private static class IntegerCache {
    static final int low = -128;
    static final int high;
    static final Integer cache[];

    static {
        // high value may be configured by property
        int h = 127;
        String integerCacheHighPropValue =
            sun.misc.VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
        if (integerCacheHighPropValue != null) {
            try {
                int i = parseInt(integerCacheHighPropValue);
                i = Math.max(i, 127);
                // Maximum array size is Integer.MAX_VALUE
                h = Math.min(i, Integer.MAX_VALUE - (-low) -1);
            } catch( NumberFormatException nfe) {
                // If the property cannot be parsed into an int, ignore it.
            }
        }
        high = h;

        cache = new Integer[(high - low) + 1];
        int j = low;
        for(int k = 0; k < cache.length; k++)
            cache[k] = new Integer(j++);

        // range [-128, 127] must be interned (JLS7 5.1.7)
        assert IntegerCache.high >= 127;
    }

    private IntegerCache() {}
}
```

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

提醒：越是貌似简单的面试题其中的玄机就越多，需要面试者有相当深厚的功力。

8、&和&&的区别？
答：&运算符有两种用法：(1)按位与；(2)逻辑与。&&运算符是短路与运算。逻辑与跟短路与的差别是非常巨大的，虽然二者都要求运算符左右两端的布尔值都是 true 整个表达式的值才是 true。&&之所以称为短路运算是因为，如果&&左边的表达式的值是 false，右边的表达式会被直接短路掉，不会进行运算。很多时候我们可能都需要用&&而不是&，例如在验证用户登录时判定用户名不是 null 而且不是空字符串，应当写为：username != null &&!username.equals("")，二者的顺序不能交换，更不能用&运算符，因为第一个条件如果不成立，根本不能进行字符串的 equals 比较，否则会产生 NullPointerException 异常。注意：逻辑或运算符（|）和短路或运算符（||）的差别也是如此。

补充：如果你熟悉 JavaScript，那你可能更能感受到短路运算的强大，想成为 JavaScript 的高手就先从玩转短路运算开始吧。

9、解释内存中的栈(stack)、堆(heap)和静态区(static area)的用法。
答：通常我们定义一个基本数据类型的变量，一个对象的引用，还有就是函数调用的现场保存都使用内存中的栈空间；而通过 new 关键字和构造器创建的对象放在堆空间；程序中的字面量（literal）如直接书写的 100、"hello"和常量都是放在静态区中。栈空间操作起来最快但是栈很小，通常大量的对象都是放在堆空间，理论上整个内存没有被其他进程使用的空间甚至硬盘上的虚拟内存都可以被当成堆空间来使用。

String str = new String("hello");

上面的语句中变量 str 放在栈上，用 new 创建出来的字符串对象放在堆上，而"hello"这个字面量放在静态区。

补充：较新版本的 Java（从 Java 6 的某个更新开始）中使用了一项叫"逃逸分析"的技术，可以将一些局部对象放在栈上以提升对象的操作性能。

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

11、swtich 是否能作用在 byte 上，是否能作用在 long 上，是否能作用在 String 上？
答：在 Java 5 以前，switch(expr)中，expr 只能是 byte、short、char、int。从 Java 5 开始，Java 中引入了枚举类型，expr 也可以是 enum 类型，从 Java 7 开始，expr 还可以是字符串（String），但是长整型（long）在目前所有的版本中都是不可以的。

12、用最有效率的方法计算 2 乘以 8？
答： 2 << 3（左移 3 位相当于乘以 2 的 3 次方，右移 3 位相当于除以 2 的 3 次方）。

补充：我们为编写的类重写 hashCode 方法时，可能会看到如下所示的代码，其实我们不太理解为什么要使用这样的乘法运算来产生哈希码（散列码），而且为什么这个数是个素数，为什么通常选择 31 这个数？前两个问题的答案你可以自己百度一下，选择 31 是因为可以用移位和减法运算来代替乘法，从而得到更好的性能。说到这里你可能已经想到了：31 * num 等价于(num << 5) - num，左移 5 位相当于乘以 2 的 5 次方再减去自身就相当于乘以 31，现在的 VM 都能自动完成这个优化。

```
public class PhoneNumber {
    private int areaCode;
    private String prefix;
    private String lineNumber;

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + areaCode;
        result = prime * result
                + ((lineNumber == null) ? 0 : lineNumber.hashCode());
        result = prime * result + ((prefix == null) ? 0 : prefix.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
```

```
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        PhoneNumber other = (PhoneNumber) obj;
        if (areaCode != other.areaCode)
            return false;
        if (lineNumber == null) {
            if (other.lineNumber != null)
                return false;
        } else if (!lineNumber.equals(other.lineNumber))
            return false;
        if (prefix == null) {
            if (other.prefix != null)
                return false;
        } else if (!prefix.equals(other.prefix))
            return false;
        return true;
    }

}
```

13、数组有没有 length()方法？String 有没有 length()方法？
答：数组没有 length()方法，有 length 的属性。String 有 length()方法。JavaScript 中，获得字符串的长度是通过 length 属性得到的，这一点容易和 Java 混淆。

14、在 Java 中，如何跳出当前的多重嵌套循环？
答：在最外层循环前加一个标记(Label)如 A，然后用 break A;可以跳出多重循环。（Java 中支持带标签的 break 和 continue 语句，作用有点类似于 C 和 C++中的 goto 语句，但是就像要避免使用 goto 一样，应该避免使用带标签的 break 和 continue，因为它不会让你的程序变得更优雅，很多时候甚至有相反的作用，所以这种语法其实不知道更好）

15、构造器（constructor）是否可被重写（override）？
答：构造器不能被继承，因此不能被重写，但可以被重载。

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

补充：关于 equals 和 hashCode 方法，很多 Java 程序都知道，但很多人也就是仅仅知道而已，在 Joshua Bloch 的大作《Effective Java》（很多软件公司，《Effective Java》、《Java 编程思想》以及《重构：改善既有代码质量》是 Java 程序员必看书籍，如果你还没看过，那就赶紧去亚马逊买一本吧）中是这样介绍 equals 方法的：首先 equals 方法必须满足自反性（x.equals(x)必须返回 true）、对称性（x.equals(y)返回 true 时，y.equals(x)也必须返回 true）、传递性（x.equals(y)和 y.equals(z)都返回 true 时，x.equals(z)也必须返回 true）和一致性（当 x 和 y 引用的对象信息没有被修改时，多次调用 x.equals(y)应该得到同样的返回值），而且对于任何非 null 值的引用 x，x.equals(null)必须返回 false。

实现高质量的equals方法的诀窍包括：1. 使用==操作符检查"参数是否为这个对象的引用"；

2. 使用instanceof操作符检查"参数是否为正确的类型"；3. 对于类中的关键属性，检查参数传入对象的属性是否与之相匹配；4. 编写完equals方法后，问自己它是否满足对称性、传递性、一致性；5. 重写equals时总是要重写hashCode；6. 不要将equals方法参数中的Object对象替换为其他的类型，在重写时不要忘掉@Override注解。

17、是否可以继承String类？
答：String 类是final类，不可以被继承。

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

18、当一个对象被当作参数传递到一个方法后，此方法可改变这个对象的属性，并可返回变化后的结果，那么这里到底是值传递还是引用传递？
答：是值传递。Java语言的方法调用只支持参数的值传递。当一个对象实例作为一个参数被传递到方法中时，参数的值就是对该对象的引用。对象的属性可以在被调用过程中被改变，但对对象引用的改变是不会影响到调用者的。C++和C#中可以通过传引用或传输出参数来改变传入的参数的值。在C#中可以编写如下所示的代码，但是在Java中却做不到。

```
using System;

namespace CS01 {

  class Program {
    public static void swap(ref int x, ref int y) {
      int temp = x;
      x = y;
      y = temp;
    }

    public static void Main (string[] args) {
      int a = 5, b = 10;
      swap (ref a, ref b);
      // a = 10, b = 5;
      Console.WriteLine ("a = {0}, b = {1}", a, b);
    }
  }
}
```

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

19、String和StringBuilder、StringBuffer的区别？
答：Java平台提供了两种类型的字符串：String和StringBuffer/StringBuilder，它们可以储存和操作字符串。其中String是只读字符串，也就意味着String引用的字符串内容是不能被改变的。而StringBuffer/StringBuilder类表示的字符串对象可以直接进行修改。StringBuilder是Java 5中引入的，它和StringBuffer的方法完全相同，区别在于它是在单线程环境下使用的，因为它的所有方面都没有被synchronized修饰，因此它的效率也比StringBuffer要高。

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

如果是少量的字符串拼接，可以用+，如果是大量的还是用StringBuffer/StringBuilder吧。StringBuffer是线程安全的，StringBuilder是线程不安全的，很

明显，StringBuffer的系统开销要大，所以如果我们只有一个单线程，考虑速度的话，StringBuilder更好。 那为什么我们很少见到StringBuilder呢？原因很简单，因为我们有时候很难确定我们创建的系统会不会是多线程的，如果考虑到以后扩展的可能性，则更难确定，所以我们更愿意使用StringBuffer，因为它是线程安全的，不用担心以后扩展。

面试题2 - 请说出下面程序的输出。

```
class StringEqualTest {

   public static void main(String[] args) {
      String s1 = "Programming";
      String s2 = new String("Programming");
      String s3 = "Program" + "ming";
      System.out.println(s1 == s2);
      System.out.println(s1 == s3);
      System.out.println(s1 == s2.intern());
            System.out.println(s2.intern() == "Programming");
   }
}
false
true
true
true
```

补充：解答上面的面试题需要清楚两点：1. String 对象的 intern 方法会得到字符串对象在常量池中对应的版本的引用（如果常量池中有一个字符串与 String 对象的 equals 结果是 true），如果常量池中没有对应的字符串，则该字符串将被添加到常量池中，然后返回常量池中字符串的引用；2. 字符串的+操作其本质是创建了 StringBuilder 对象进行 append 操作，然后将拼接后的 StringBuilder 对象用 toString 方法处理成 String 对象，这一点可以用 javap -c StringEqualTest.class 命令获得 class 文件对应的 JVM 字节码指令就可以看出来。String.intern()方法就是把该对象放进常量池。

20、重载（Overload）和重写（Override）的区别。重载的方法能否根据返回类型进行区分？
答：方法的重载和重写都是实现多态的方式，区别在于前者实现的是编译时的多态性，而后者实现的是运行时的多态性。重载发生在一个类中，同名的方法如果有不同的参数列表（参数类型不同、参数个数不同或者二者都不同）则视为重载；重写发生在子类与父类之间，重写要求子类被重写方法与父类被重写方法有相同的返回类型，比父类被重写方法更好访问，不能比父类被重写方法声明更多的异常（里氏代换原则）。重载对返回类型没有特殊的要求。

面试题：华为的面试题中曾经问过这样一个问题 - "为什么不能根据返回类型来区分重载"，快说出你的答案吧！

返回类型不同不构成重载，重载的机制是参数列表不同，即参数的类型，个数，排列顺序。

21、描述一下 JVM 加载 class 文件的原理机制？
答：JVM 中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java 中的类加载器是一个重要的 Java 运行时系统组件，它负责在运行时查找和装入类文件中的类。
由于 Java 的跨平台性，经过编译的 Java 源程序并不是一个可执行程序，而是一个或多个类文件。当 Java 程序需要使用某个类时，JVM 会确保这个类已经被加载、连接（验证、准备和解析）和初始化。类的加载是指把类的.class 文件中的数据读入到内存中，通常是创建一个字节数组读入.class 文件，然后产生与所加载类对应的 Class 对象。加载完成后，Class 对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、

准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后 JVM 对类进行初始化，包括：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。
类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader 的子类）。从 Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。PDM 更好的保证了 Java 平台的安全性，在该机制中，JVM 自带的 Bootstrap 是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM 不会向 Java 程序提供对 Bootstrap 的引用。下面是关于几个类加载器的说明：

Bootstrap：一般用本地代码实现，负责加载 JVM 基础核心类库（rt.jar）；
Extension：从 java.ext.dirs 系统属性所指定的目录中加载类库，它的父加载器是 Bootstrap；
System：又叫应用类加载器，其父类是 Extension。它是应用最广泛的类加载器。它从环境变量 classpath 或者系统属性 java.class.path 所指定的目录中记载类，是用户自定义加载器的默认父加载器。

22、char 型变量中能不能存贮一个中文汉字，为什么？
答：char 类型可以存储一个中文汉字，因为 Java 中使用的编码是 Unicode（不选择任何特定的编码，直接使用字符在字符集中的编号，这是统一的唯一方法），一个 char 类型占 2 个字节（16 比特），所以放一个中文是没问题的。

补充：使用 Unicode 意味着字符在 JVM 内部和外部有不同的表现形式，在 JVM 内部都是 Unicode，当这个字符被从 JVM 内部转移到外部时（例如存入文件系统中），需要进行编码转换。所以 Java 中有字节流和字符流，以及在字符流和字节流之间进行转换的转换流，如 InputStreamReader 和 OutputStreamReader，这两个类是字节流和字符流之间的适配器类，承担了编码转换的任务；对于 C 程序员来说，要完成这样的编码转换恐怕要依赖于 union（联合体/共用体）共享内存的特征来实现了。

23、抽象类（abstract class）和接口（interface）有什么异同？
答：抽象类和接口都不能够实例化，但可以定义抽象类和接口类型的引用。一个类如果继承了某个抽象类或者实现了某个接口都需要对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。接口比抽象类更加抽象，因为抽象类中可以定义构造器，可以有抽象方法和具体方法，而接口中不能定义构造器而且其中的方法全部都是抽象方法。抽象类中的成员可以是 private、默认、protected、public 的，而接口中的成员全都是 public 的。抽象类中可以定义成员变量，而接口中定义的成员变量实际上都是常量。有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法。

24、静态嵌套类(Static Nested Class)和内部类（Inner Class）的不同？
答：Static Nested Class 是被声明为静态（static）的内部类，它可以不依赖于外部类实例被实例化。而通常的内部类需要在外部类实例化后才能实例化，其语法看起来挺诡异的，如下所示。

```
/**
 * 扑克类（一副扑克）
 * @author 骆昊
 *
 */
public class Poker {
    private static String[] suites = {"黑桃", "红桃", "草花", "方块"};
```

```
private static int[] faces = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13};

private Card[] cards;

/**
 * 构造器
 *
 */
public Poker() {
    cards = new Card[52];
    for(int i = 0; i < suites.length; i++) {
        for(int j = 0; j < faces.length; j++) {
            cards[i * 13 + j] = new Card(suites[i], faces[j]);
        }
    }
}

/**
 * 洗牌 （随机乱序）
 *
 */
public void shuffle() {
    for(int i = 0, len = cards.length; i < len; i++) {
        int index = (int) (Math.random() * len);
        Card temp = cards[index];
        cards[index] = cards[i];
        cards[i] = temp;
    }
}

/**
 * 发牌
 * @param index 发牌的位置
 *
 */
public Card deal(int index) {
    return cards[index];
}

/**
 * 卡片类（一张扑克）
 * [内部类]
 * @author 骆昊
 *
 */
public class Card {
    private String suite;   // 花色
    private int face;       // 点数

    public Card(String suite, int face) {
        this.suite = suite;
        this.face = face;
    }

    @Override
    public String toString() {
```

```
            String faceStr = "";
            switch(face) {
            case 1: faceStr = "A"; break;
            case 11: faceStr = "J"; break;
            case 12: faceStr = "Q"; break;
            case 13: faceStr = "K"; break;
            default: faceStr = String.valueOf(face);
            }
            return suite + faceStr;
        }
    }
}
```

测试代码：

```
class PokerTest {

    public static void main(String[] args) {
        Poker poker = new Poker();
        poker.shuffle();              // 洗牌
        Poker.Card c1 = poker.deal(0);  // 发第一张牌
        // 对于非静态内部类 Card
        // 只有通过其外部类 Poker 对象才能创建 Card 对象
        Poker.Card c2 = poker.new Card("红心", 1);    // 自己创建一张牌

        System.out.println(c1);     // 洗牌后的第一张
        System.out.println(c2);     // 打印: 红心 A
    }
}
```

面试题 - 下面的代码哪些地方会产生编译错误？

```
class Outer {

    class Inner {}

    public static void foo() { new Inner(); }

    public void bar() { new Inner(); }

    public static void main(String[] args) {
        new Inner();
    }
}
```

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

```
new Outer().new Inner();
```

25、Java 中会存在内存泄漏吗，请简单描述。
答：理论上 Java 因为有垃圾回收机制（GC）不会存在内存泄露问题（这也是 Java 被广泛使用于服务器端编程的一个重要原因）；然而在实际开发中，可能会存在无用但可达的对象，这些对象不能被 GC 回收，因此也会导致内存泄露的发生。例如 Hibernate 的 Session

（一级缓存）中的对象属于持久态，垃圾回收器是不会回收这些对象的，然而这些对象中可能存在无用的垃圾对象，如果不及时关闭（close）或清空（flush）一级缓存就可能导致内存泄露。下面例子中的代码也会导致内存泄露。

```
import java.util.Arrays;
import java.util.EmptyStackException;

public class MyStack<T> {
   private T[] elements;
   private int size = 0;

   private static final int INIT_CAPACITY = 16;

   public MyStack() {
      elements = (T[]) new Object[INIT_CAPACITY];
   }

   public void push(T elem) {
      ensureCapacity();
      elements[size++] = elem;
   }

   public T pop() {
      if(size == 0)
         throw new EmptyStackException();
      return elements[--size];
   }

   private void ensureCapacity() {
      if(elements.length == size) {
         elements = Arrays.copyOf(elements, 2 * size + 1);
      }
   }
}
```

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

26、抽象的（abstract）方法是否可同时是静态的（static）,是否可同时是本地方法（native），是否可同时被 synchronized 修饰？
答：都不能。抽象方法需要子类重写，而静态的方法是无法被重写的，因此二者是矛盾的。本地方法是由本地代码（如 C 代码）实现的方法，而抽象方法是没有实现的，也是矛盾的。synchronized 和方法的实现细节有关，抽象方法不涉及实现细节，因此也是相互矛盾的。

27、阐述静态变量和实例变量的区别。
答：静态变量是被 static 修饰符修饰的变量，也称为类变量，它属于类，不属于类的任何一个对象，一个类不管创建多少个对象，静态变量在内存中有且仅有一个拷贝；实例变量必须依存于某一实例，需要先创建对象然后通过对象才能访问到它。静态变量可以实现让

多个对象共享内存。

补充：在 Java 开发中，上下文类和工具类中通常会有大量的静态成员。

28、是否可以从一个静态（static）方法内部发出对非静态（non-static）方法的调用？
答：不可以，静态方法只能访问静态成员，因为非静态方法的调用要先创建对象，在调用静态方法时可能对象并没有被初始化。

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

```
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

public class MyUtil {

    private MyUtil() {
        throw new AssertionError();
    }

    public static <T> T clone(T obj) throws Exception {
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bout);
        oos.writeObject(obj);

        ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bin);
        return (T) ois.readObject();

        // 说明：调用 ByteArrayInputStream 或 ByteArrayOutputStream 对象的 close 方法没有任何意义
        // 这两个基于内存的流只要垃圾回收器清理对象就能够释放资源，这一点不同于对外部资源（如文件流）的释放
    }
}
```

下面是测试代码：

```
import java.io.Serializable;

/**
 * 人类
 * @author 骆昊
 *
 */
class Person implements Serializable {
    private static final long serialVersionUID = -9102017020286042305L;

    private String name;    // 姓名
    private int age;        // 年龄
```

```
    private Car car;        // 座驾

    public Person(String name, int age, Car car) {
        this.name = name;
        this.age = age;
        this.car = car;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Car getCar() {
        return car;
    }

    public void setCar(Car car) {
        this.car = car;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + ", car=" + car + "]";
    }

}



/**
 * 小汽车类
 * @author 骆昊
 *
 */
class Car implements Serializable {
    private static final long serialVersionUID = -5713945027627603702L;

    private String brand;       // 品牌
    private int maxSpeed;       // 最高时速

    public Car(String brand, int maxSpeed) {
        this.brand = brand;
        this.maxSpeed = maxSpeed;
    }
```

```
    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }

    public int getMaxSpeed() {
        return maxSpeed;
    }

    public void setMaxSpeed(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    @Override
    public String toString() {
        return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]";
    }

}

class CloneTest {
        public static void main(String[] args) {
                try {
                        Person p1 = new Person("Hao LUO", 33, new Car("Benz",
300));
                        Person p2 = MyUtil.clone(p1);   // 深度克隆
                        p2.getCar().setBrand("BYD");
                        // 修改克隆的 Person 对象 p2 关联的汽车对象的品牌属
性
                    // 原来的 Person 对象 p1 关联的汽车不会受到任何影响
                        // 因为在克隆 Person 对象时其关联的汽车对象也被克隆
了
                        System.out.println(p1);
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

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

30、GC 是什么？为什么要有 GC？
答：GC 是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java 提供的 GC 功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java 语言没有提供释放已分配内存的显示操作方法。Java 程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc() 或 Runtime.getRuntime().gc() ，但 JVM 可以屏蔽掉显示的垃圾回收调用。
垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为

一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。在 Java 诞生初期，垃圾回收是 Java 最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今 Java 的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得 iOS 的系统比 Android 系统有更好的用户体验，其中一个深层次的原因就在于 Android 系统中垃圾回收的不可预知性。

补充：垃圾回收机制有很多种，包括：分代复制垃圾回收、标记垃圾回收、增量垃圾回收等方式。标准的 Java 进程既有栈又有堆。栈保存了原始型局部变量，堆保存了要创建的对象。Java 平台对堆内存回收和再利用的基本算法被称为标记和清除，但是 Java 对其进行了改进，采用“分代式垃圾收集”。这种方法会跟 Java 对象的生命周期将堆内存划分为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：

- 伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。
- 幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。
- 终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。

与垃圾回收相关的 JVM 参数：

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

31、String s = new String("xyz");创建了几个字符串对象？
答：一个或两个对象，如果常量池中已经有"xyz"，则只需要创建 new 在堆上的对象；如果没有，则需要创建两个对象，一个是静态区的"xyz"，一个是用 new 创建在堆上的对象。

32、接口是否可继承（extends）接口？抽象类是否可实现（implements）接口？抽象类是否可继承具体类（concrete class）？
答：接口可以继承接口，而且支持多重继承。抽象类可以实现(implements)接口，抽象类可继承具体类也可以继承抽象类。
当你自己写的类想用接口中个别方法的时候（注意不是所有的方法），那么你就可以用一个抽象类先实现这个接口（方法体中为空），然后再用你的类继承这个抽象类，这样就可以达到你的目的了，如果你直接用类实现接口，那是所有方法都必须实现的。
33、一个".java"源文件中是否可以包含多个类（不是内部类）？有什么限制？
答：可以，但一个源文件中最多只能有一个公开类（public class）而且文件名必须和公开类的类名完全保持一致。

34、Anonymous Inner Class(匿名内部类)是否可以继承其它类？是否可以实现接口？
答：可以继承其他类或实现其他接口，在 Swing 编程和 Android 开发中常用此方式来实现

事件监听和回调。
匿名内部类在实现时必须借助一个借口或者一个抽象类或者一个普通类来构造，从这过层次上讲匿名内部类是实现了接口或者继承了类，但是不能通过 extends 或 implement 关键词来继承类或实现接口。
匿名内部类即没有名字的内部类。当我们只需要用某一个类一次时，且该类从意义上需要实现某个类或某个接口，这个特殊的扩展类就以匿名内部类来展现。
一般的用途：
1、覆盖某个超类的方法，并且该扩展类只在本类内用一次。
2、继承抽象类，并实例化其抽象方法，并且该扩展类只在本类内用一次。
3、实现接口，实例化其方法，并且该扩展类只在本类内用一次。
代码示例：

```
class Car{
   void move(){}
}
interface Person{
   void learn();
}
abstract class Animal{
   abstract void eat();
}
public class AnonymousInnerClassDemo {

   public static void main(String[] args) {
      Car car = new Car(){
         @Override
         void move() {
            System.out.println("匿名内部类的 move 方法");
         }
      };
      car.move();

      Person person = new Person() {

         public void learn() {
            System.out.println("匿名内部类的 learn 方法");
         }
      };
      person.learn();

      Animal animal = new Animal() {
         @Override
         void eat() {
            System.out.println("匿名内部类的 eat 方法");
         }
      };
      animal.eat();
   }
}
```

几点说明：
一、由于匿名内部类没有名字，所以它没有构造函数。因为没有构造函数，所以它必须完全借用父类的构造函数来实例化，匿名内部类完全把创建对象的任务交给了父类去完成。
二、在匿名内部类里创建新的方法没有太大意义，但它可以通过覆盖父类的方法达到神奇效果，如上例所示。这是多态性的体现。
三、因为匿名内部类没有名字，所以无法进行向下的强制类型转换，持有对一个匿名内部类对象引用的变量类型一定是它的直接或间接父类类型。

四、注意匿名内部类的声明是在编译时进行的，实例化在运行时进行。这意味着 for 循环中的一个 new 语句会创建相同匿名类的几个实例，而不是创建几个不同匿名类的一个实例。

35、内部类可以引用它的包含类（外部类）的成员吗？有没有什么限制？
答：一个内部类对象可以访问创建它的外部类对象的成员，包括私有成员。

36、Java 中的 final 关键字有哪些用法？
答：(1)修饰类：表示该类不能被继承；(2)修饰方法：表示方法不能被重写；(3)修饰变量：表示变量只能一次赋值以后值不能被修改（常量）。

37、指出下面程序的运行结果。

```
class A {

   static {
      System.out.print("1");
   }

   public A() {
      System.out.print("2");
   }
}

class B extends A{

   static {
      System.out.print("a");
   }

   public B() {
      System.out.print("b");
   }
}

public class Hello {

   public static void main(String[] args) {
      A ab = new B();
      ab = new B();                    //静态代码块只会初始化一次
   }

}
```

答：执行结果：1a2b2b。创建对象时构造器的调用顺序是：先初始化静态成员，然后调用父类构造器，再初始化非静态成员，最后调用自身构造器。
提示：如果不能给出此题的正确答案，说明之前第 21 题 Java 类加载机制还没有完全理解，赶紧再看看吧。

38、数据类型之间的转换：
- 如何将字符串转换为基本数据类型？
- 如何将基本数据类型转换为字符串？
答：
- 调用基本数据类型对应的包装类中的方法 parseXXX(String)或 valueOf(String)即可返回相应基本类型；parseXXX(String)返回值是基本数据类型，valueOf(String)返回类型是对应的封装类型。

```
String string="123";
int num=Integer.parseInt(string);
float num2=Floar.parseFloat(string);
double num3=Double.parseDouble(string);
```
其中，parseXXX 用于把 String 类型的变量转换为基本类型(比如 int float double)

```
Integer integer= Integer.valueOf(123);
Double  dou= Double.valueOf(123);
```
valueOf( )是一种生成对应基本类型的包装类的一种方法

- 一种方法是将基本数据类型与空字符串（""）连接（+）即可获得其所对应的字符串；另一种方法是调用 String 类中的 valueOf()方法返回相应字符串

39、如何实现字符串的反转及替换？
答：方法很多，可以自己写实现也可以使用 String 或 StringBuffer/StringBuilder 中的方法。有一道很常见的面试题是用递归实现字符串反转，代码如下所示：

```
  public static String reverse(String originStr) {
      if(originStr == null || originStr.length() <= 1)
         return originStr;
             //递归调用，substring(1)，截取从索引1开始到后面的字符串，charAt(0)返回指定索引0的字符
      return reverse(originStr.substring(1)) + originStr.charAt(0);
   }
```

40、怎样将GB2312编码的字符串转换为ISO-8859-1编码的字符串？
答：代码如下所示：

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

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

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

```
public class DateTimeTest {
   public static void main(String[] args) {
      Calendar cal = Calendar.getInstance();
      System.out.println(cal.get(Calendar.YEAR));
      System.out.println(cal.get(Calendar.MONTH));    // 0 - 11
      System.out.println(cal.get(Calendar.DATE));
      System.out.println(cal.get(Calendar.HOUR_OF_DAY));
      System.out.println(cal.get(Calendar.MINUTE));
      System.out.println(cal.get(Calendar.SECOND));

      // Java 8
      LocalDateTime dt = LocalDateTime.now();
      System.out.println(dt.getYear());
      System.out.println(dt.getMonthValue());     // 1 - 12
      System.out.println(dt.getDayOfMonth());
```

```
        System.out.println(dt.getHour());
        System.out.println(dt.getMinute());
        System.out.println(dt.getSecond());
    }
}
```

问题2：以下方法均可获得该毫秒数。

```
Calendar.getInstance().getTimeInMillis();
System.currentTimeMillis();
Clock.systemDefaultZone().millis(); // Java 8
```

问题3：代码如下所示。

```
Calendar time = Calendar.getInstance();
time.getActualMaximum(Calendar.DAY_OF_MONTH);
```

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

```
import java.text.SimpleDateFormat;
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.Date;

class DateFormatTest {

    public static void main(String[] args) {
        SimpleDateFormat oldFormatter = new SimpleDateFormat("yyyy/MM/dd");
        Date date1 = new Date();
        System.out.println(oldFormatter.format(date1));

        // Java 8
        DateTimeFormatter newFormatter = DateTimeFormatter.ofPattern("yyyy/MM/dd");
        LocalDate date2 = LocalDate.now();
        System.out.println(date2.format(newFormatter));
    }
}
```

补充：Java的时间日期API一直以来都是被诟病的东西，为了解决这一问题，Java 8中引入了新的时间日期API，其中包括LocalDate、LocalTime、LocalDateTime、Clock、Instant等类，这些的类的设计都使用了不变模式，因此是线程安全的设计。

42、打印昨天的当前时刻。
答：

```
import java.util.Calendar;

class YesterdayCurrent {
    public static void main(String[] args){
        Calendar cal = Calendar.getInstance();
        cal.add(Calendar.DATE, -1);
        System.out.println(cal.getTime());
    }
}
```

在Java 8中，可以用下面的代码实现相同的功能。

```
import java.time.LocalDateTime;

class YesterdayCurrent {

   public static void main(String[] args) {
      LocalDateTime today = LocalDateTime.now();
      LocalDateTime yesterday = today.minusDays(1);

      System.out.println(yesterday);
   }
}
```

43、比较一下Java和JavaSciprt。
答：JavaScript 与Java是两个公司开发的不同的两个产品。Java 是原Sun Microsystems公司推出的面向对象的程序设计语言，特别适合于互联网应用程序开发；而JavaScript是Netscape公司的产品，为了扩展Netscape浏览器的功能而开发的一种可以嵌入Web页面中运行的基于对象和事件驱动的解释性语言。JavaScript的前身是LiveScript；而Java的前身是Oak语言。
下面对两种语言间的异同作如下比较：
- 基于对象和面向对象：Java是一种真正的面向对象的语言，即使是开发简单的程序，必须设计对象；JavaScript是种脚本语言，它可以用来制作与网络无关的，与用户交互作用的复杂软件。它是一种基于对象（Object-Based）和事件驱动（Event-Driven）的编程语言，因而它本身提供了非常丰富的内部对象供设计人员使用。
- 解释和编译：Java的源代码在执行之前，必须经过编译。JavaScript是一种解释性编程语言，其源代码不需经过编译，由浏览器解释执行。（目前的浏览器几乎都使用了JIT（即时编译）技术来提升JavaScript的运行效率）
- 强类型变量和类型弱变量：Java采用强类型变量检查，即所有变量在编译之前必须作声明；JavaScript中变量是弱类型的，甚至在使用变量前可以不作声明，JavaScript的解释器在运行时检查推断其数据类型。
- 代码格式不一样。

补充：上面列出的四点是网上流传的所谓的标准答案。其实Java和JavaScript最重要的区别是一个是静态语言，一个是动态语言。目前的编程语言的发展趋势是函数式语言和动态语言。在Java中类（class）是一等公民，而JavaScript中函数（function）是一等公民，因此JavaScript支持函数式编程，可以使用Lambda函数和闭包（closure），当然Java 8也开始支持函数式编程，提供了对Lambda表达式以及函数式接口的支持。对于这类问题，在面试的时候最好还是用自己的语言回答会更加靠谱，不要背网上所谓的标准答案。

44、什么时候用断言（assert）？
答：断言在软件开发中是一种常用的调试方式，很多开发语言中都支持这种机制。一般来说，断言用于保证程序最基本、关键的正确性。断言检查通常在开发和测试时开启。为了保证程序的执行效率，在软件发布后断言检查通常是关闭的。断言是一个包含布尔表达式的语句，在执行这个语句时假定该表达式为true；如果表达式的值为false，那么系统会报告一个AssertionError。断言的使用如下面的代码所示：

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

断言可以有两种形式：
assert Expression1;
assert Expression1 : Expression2 ;
Expression1 应该总是产生一个布尔值。

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

要在运行时启用断言，可以在启动JVM时使用-enableassertions或者-ea标记。要在运行时选择禁用断言，可以在启动JVM时使用-da或者-disableassertions标记。要在系统类中启用或禁用断言，可使用-esa或-dsa标记。还可以在包的基础上启用或者禁用断言。

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

45、Error和Exception有什么区别？
答：Error表示系统级的错误和程序不必处理的异常，是恢复不是不可能但很困难的情况下的一种严重问题；比如内存溢出，不可能指望程序能处理这样的情况；Exception表示需要捕捉或者需要程序进行处理的异常，是一种设计或实现问题；也就是说，它表示如果程序运行正常，从不会发生的情况。

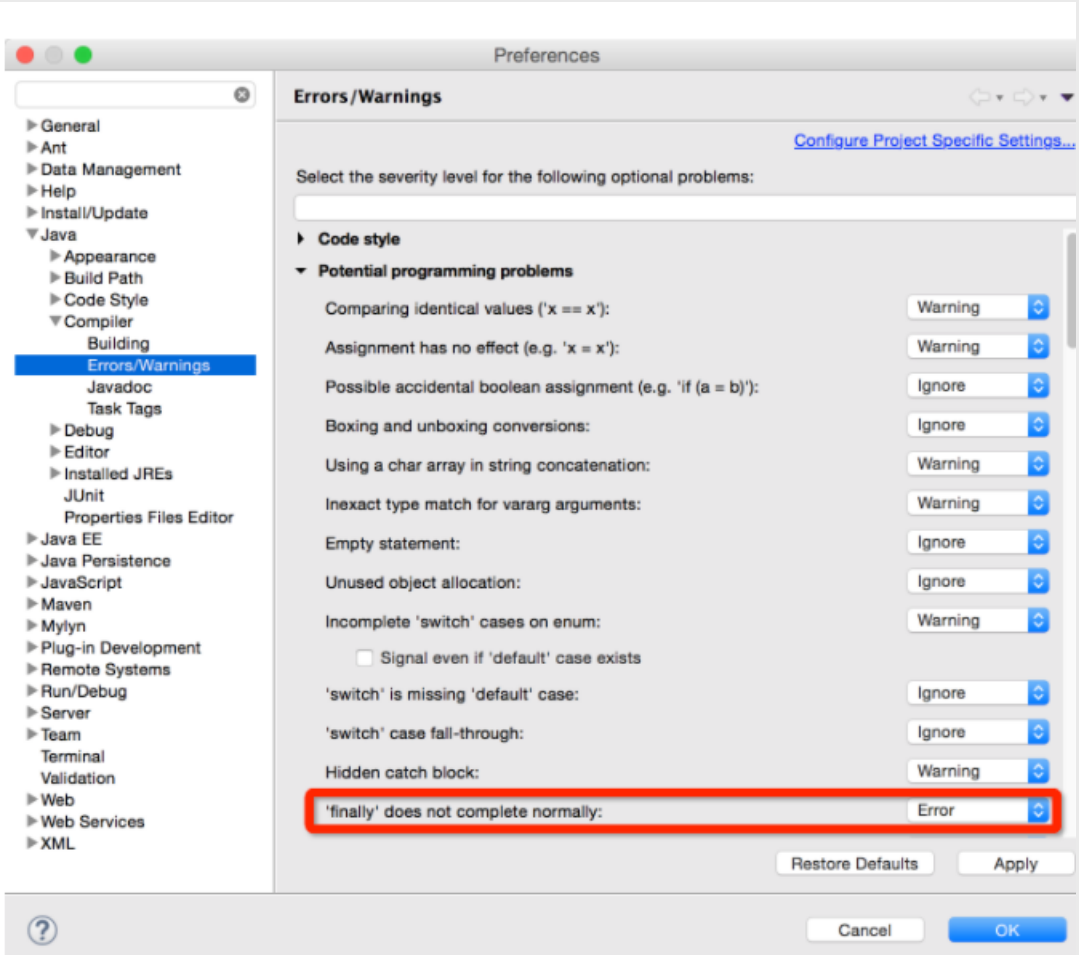
面试题：2005年摩托罗拉的面试中曾经问过这么一个问题“If a process reports a stack overflow run-time error, what’s the most possible cause?”，给了四个选项a. lack of memory; b. write on an invalid memory space; c. recursive function calling; d. array index out of boundary. Java程序在运行时也可能会遭遇StackOverflowError，这是一个无法恢复的错误，只能重新修改代码了，这个面试题的答案是c。如果写了不能迅速收敛的递归，则很有可能引发栈溢出的错误，如下所示：

```
class StackOverflowErrorTest {

   public static void main(String[] args) {
      main(null);
   }
}
```

提示：用递归编写程序时一定要牢记两点：1. 递归公式；2. 收敛条件（什么时候就不再继续递归）。

46、try{}里有一个return语句，那么紧跟在这个try后的finally{}里的代码会不会被执行，什么时候被执行，在return前还是后?
答：会执行，在方法返回调用者前执行。

注意：在finally中改变返回值的做法是不好的，因为如果存在finally代码块，try中的return语句不会立马返回调用者，而是记录下返回值待finally代码块执行完毕之后再向调用者返回其值，然后如果在finally中修改了返回值，就会返回修改后的值。显然，在finally中返回或者修改返回值会对程序造成很大的困扰，C#中直接用编译错误的方式来阻止程序员干这种龌龊的事情，Java中也可以通过提升编译器的语法检查级别来产生警告或错误，Eclipse中可以在如图所示的地方进行设置，强烈建议将此项设置为编译错误。



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

答：Java通过面向对象的方法进行异常处理，把各种不同的异常进行分类，并提供了良好的接口。在Java中，每个异常都是一个对象，它是Throwable类或其子类的实例。当一个方法出现异常后便抛出一个异常对象，该对象中包含有异常信息，调用这个对象的方法可以捕获到这个异常并可以对其进行处理。Java的异常处理是通过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，如果系统会抛出（throw）一个异常对象，可以通过它的类型来捕获（catch）它，或通过总是执行代码块（finally）来处理；try用来指定一块预防所有异常的程序；catch子句紧跟在try块后面，用来指定你想要捕获的异常的类型；throw语句用来明确地抛出一个异常；throws用来声明一个方法可能抛出的各种异常（当然声明异常时允许无病呻吟）；finally为确保一段代码不管发生什么异常状况都要被执行；try语句可以嵌套，每当遇到一个try语句，异常的结构就会被放入异常栈中，直到所有的try语句都完成。如果下一级的try语句没有对某种异常进行处理，异常栈就会执行出栈操作，直到遇到有处理这种异常的try语句或者最终将异常抛给JVM。

48、运行时异常与受检异常有何异同？

答：异常表示程序运行过程中可能出现的非正常状态，运行时异常表示虚拟机的通常操作中可能遇到的异常，是一种常见运行错误，只要程序设计得没有问题通常就不会发生。受检异常跟程序运行的上下文环境有关，即使程序设计无误，仍然可能因使用的问题而引发。Java编译器要求方法必须声明抛出可能发生的受检异常，但是并不要求必须声明抛出未被捕获的运行时异常。异常和继承一样，是面向对象程序设计中经常被滥用的东西，在Effective Java中对异常的使用给出了以下指导原则：

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

制流而使用异常）
- 对可以恢复的情况使用受检异常，对编程错误使用运行时异常
- 避免不必要的使用受检异常（可以通过一些状态检测手段来避免异常的发生）
- 优先使用标准的异常
- 每个方法抛出的异常都要有文档
- 保持异常的原子性
- 不要在catch中忽略掉捕获到的异常

49、列出一些你常见的运行时异常？
答：
- ArithmeticException（算术异常）
- ClassCastException （类转换异常）
- IllegalArgumentException （非法参数异常）
- IndexOutOfBoundsException （下标越界异常）
- NullPointerException （空指针异常）
- SecurityException （安全异常）

50、阐述final、finally、finalize的区别。
答：
- final：修饰符（关键字）有三种用法：如果一个类被声明为final，意味着它不能再派生出新的子类，即不能被继承，因此它和abstract是反义词。将变量声明为final，可以保证它们在使用中不被改变，被声明为final的变量必须在声明时给定初值，而在以后的引用中只能读取不可修改。被声明为final的方法也同样只能使用，不能在子类中被重写。
- finally：通常放在try…catch…的后面构造总是执行代码块，这就意味着程序无论正常执行还是发生异常，这里的代码只要JVM不关闭都能执行，可以将释放外部资源的代码写在finally块中。
- finalize：Object类中定义的方法，Java中允许使用finalize()方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。这个方法是由垃圾收集器在销毁对象时调用的，通过重写finalize()方法可以整理系统资源或者执行其他清理工作。

51、类ExampleA继承Exception，类ExampleB继承ExampleA。
有如下代码片断：

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

请问执行此段代码的输出是什么？
答：输出：ExampleA。（根据里氏代换原则[能使用父类型的地方一定能使用子类型]，抓取ExampleA类型异常的catch块能够抓住try块中抛出的ExampleB类型的异常）

面试题 - 说出下面代码的运行结果。（此题的出处是《Java编程思想》一书）

```
class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

class Human {

   public static void main(String[] args)
```

```
      throws Exception {
      try {
         try {
            throw new Sneeze();
         }
         catch ( Annoyance a ) {
            System.out.println("Caught Annoyance");
            throw a;
         }
      }
      catch ( Sneeze s ) {
         System.out.println("Caught Sneeze");
         return ;
      }
      finally {
         System.out.println("Hello World!");
      }
   }
}
```

```
Caught Annoyance
Caught Sneeze
Hello World!
```

52、List、Set、Map 是否继承自 Collection 接口？

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

(1)List,Set 都是继承自 Collection 接口

(2)List 特点：元素有放入顺序，元素可重复 ，Set 特点：元素无放入顺序，元素不可重复（注意：元素虽然无放入顺序，但是元素在 set 中的位置是有该元素的 HashCode 决定的，其位置其实是固定的）

(3)List 接口有三个实现类：LinkedList，ArrayList，Vector ，Set 接口有两个实现类：HashSet(底层由 HashMap 实现)，LinkedHashSet

(4)HashMap 是支持 null 键和 null 值的，而 HashTable 在遇到 null 时，会抛出 NullPointerException 异常。这并不是因为 HashTable 有什么特殊的实现层面的原因导致不能支持 null 键和 null 值，这仅仅是因为 HashMap 在实现时对 null 做了特殊处理，将 null 的 hashCode 值定为了 0，从而将其存放在哈希表的第 0 个 bucket 中。

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

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

补充：遗留容器中的 Properties 类和 Stack 类在设计上有严重的问题，Properties 是一个键

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

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

55、List、Map、Set 三个接口存取元素时，各有什么特点？
答：List 以特定索引来存取元素，可以有重复元素。Set 不能存放重复元素（用对象的 equals()方法来区分元素是否重复）。Map 保存键值对（key-value pair）映射，映射关系可以是一对一或多对一。Set 和 Map 容器都有基于哈希存储和排序树的两种实现版本，基于哈希存储的版本理论存取时间复杂度为 O(1)，而基于排序树版本的实现在插入或删除元素时会按照元素或元素的键（key）构成排序树从而达到排序和去重的效果。

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

```
public class Student implements Comparable<Student> {
    private String name;       // 姓名
    private int age;           // 年龄

    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Student [name=" + name + ", age=" + age + "]";
    }

    @Override
    public int compareTo(Student o) {
        return this.age - o.age; // 比较年龄(年龄的升序)
    }

}
```

```
import java.util.Set;
import java.util.TreeSet;

class Test01 {

   public static void main(String[] args) {
      Set<Student> set = new TreeSet<>();     // Java 7 的钻石语法(构造器后面的尖括号中不需
要写类型)
      set.add(new Student("Hao LUO", 33));
      set.add(new Student("XJ WANG", 32));
      set.add(new Student("Bruce LEE", 60));
      set.add(new Student("Bob YANG", 22));

      for(Student stu : set) {
         System.out.println(stu);
      }
//    输出结果:
//    Student [name=Bob YANG, age=22]
//    Student [name=XJ WANG, age=32]
//    Student [name=Hao LUO, age=33]
//    Student [name=Bruce LEE, age=60]
   }
}
```

例子 2：

```
public class Student {
   private String name;    // 姓名
   private int age;        // 年龄

   public Student(String name, int age) {
      this.name = name;
      this.age = age;
   }

   /**
    * 获取学生姓名
    */
   public String getName() {
      return name;
   }

   /**
    * 获取学生年龄
    */
   public int getAge() {
      return age;
   }

   @Override
   public String toString() {
      return "Student [name=" + name + ", age=" + age + "]";
   }

}
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class Test02 {

    public static void main(String[] args) {
        List<Student> list = new ArrayList<>();     // Java 7 的钻石语法(构造器后面的尖括号中不需要写类型)
        list.add(new Student("Hao LUO", 33));
        list.add(new Student("XJ WANG", 32));
        list.add(new Student("Bruce LEE", 60));
        list.add(new Student("Bob YANG", 22));

        // 通过 sort 方法的第二个参数传入一个 Comparator 接口对象
        // 相当于是传入一个比较对象大小的算法到 sort 方法中
        // 由于 Java 中没有函数指针、仿函数、委托这样的概念
        // 因此要将一个算法传入一个方法中唯一的选择就是通过接口回调
        Collections.sort(list, new Comparator<Student> () {

            @Override
            public int compare(Student o1, Student o2) {
                return o1.getName().compareTo(o2.getName());    // 比较学生姓名
            }
        });

        for(Student stu : list) {
            System.out.println(stu);
        }
//      输出结果:
//      Student [name=Bob YANG, age=22]
//      Student [name=Bruce LEE, age=60]
//      Student [name=Hao LUO, age=33]
//      Student [name=XJ WANG, age=32]
    }
}
```

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

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

越多，程序的性能就越好，因为线程之间的调度和切换也会浪费 CPU 时间。时下很时髦的 Node.js 就采用了单线程异步 I/O 的工作模式。

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

59、当一个线程进入一个对象的 synchronized 方法 A 之后，其它线程是否可进入此对象的 synchronized 方法 B？
答：不能。其它线程只能访问该对象的非同步方法，同步方法则不能进入。因为非静态方法上的 synchronized 修饰符要求执行方法时要获得对象的锁，如果已经进入 A 方法说明对象锁已经被取走，那么试图进入 B 方法的线程就只能在等锁池（注意不是等待池哦）中等待对象的锁。

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

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

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

银行账户类：

```
/**
 * 银行账户
 * @author 骆昊
 *
 */
public class Account {
    private double balance;    // 账户余额

    /**
     * 存款
```

```
     * @param money 存入金额
     */
    public void deposit(double money) {
        double newBalance = balance + money;
        try {
            Thread.sleep(10);   // 模拟此业务需要一段处理时间
        }
        catch(InterruptedException ex) {
            ex.printStackTrace();
        }
        balance = newBalance;
    }

    /**
     * 获得账户余额
     */
    public double getBalance() {
        return balance;
    }
}
```

存钱线程类：

```
/**
 * 存钱线程
 * @author 骆昊
 *
 */
public class AddMoneyThread implements Runnable {
    private Account account;    // 存入账户
    private double money;       // 存入金额

    public AddMoneyThread(Account account, double money) {
        this.account = account;
        this.money = money;
    }

    @Override
    public void run() {
        account.deposit(money);
    }

}
```

测试类：

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Test01 {

    public static void main(String[] args) {
        Account account = new Account();
        ExecutorService service = Executors.newFixedThreadPool(100);

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

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

        service.shutdown();

        while(!service.isTerminated()) {}

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

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

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

```
/**
 * 银行账户
 * @author 骆昊
 *
 */
public class Account {
    private double balance;     // 账户余额

    /**
     * 存款
     * @param money 存入金额
     */
    public synchronized void deposit(double money) {
        double newBalance = balance + money;
        try {
            Thread.sleep(10);   // 模拟此业务需要一段处理时间
        }
        catch(InterruptedException ex) {
            ex.printStackTrace();
        }
        balance = newBalance;
    }

    /**
     * 获得账户余额
     */
    public double getBalance() {
        return balance;
    }
}
```

在线程调用存款方法时对银行账户进行同步

```
/**
```

```
 * 存钱线程
 * @author 骆昊
 *
 */
public class AddMoneyThread implements Runnable {
    private Account account;    // 存入账户
    private double money;       // 存入金额

    public AddMoneyThread(Account account, double money) {
        this.account = account;
        this.money = money;
    }

    @Override
    public void run() {
        synchronized (account) {
            account.deposit(money);
        }
    }

}
```

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

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

/**
 * 银行账户
 *
 * @author 骆昊
 *
 */
public class Account {
    private Lock accountLock = new ReentrantLock();
    private double balance; // 账户余额

    /**
     * 存款
     *
     * @param money
     *          存入金额
     */
    public void deposit(double money) {
        accountLock.lock();
        try {
            double newBalance = balance + money;
            try {
                Thread.sleep(10); // 模拟此业务需要一段处理时间
            }
            catch (InterruptedException ex) {
                ex.printStackTrace();
            }
            balance = newBalance;
        }
```

```
        finally {
            accountLock.unlock();
        }
    }

    /**
     * 获得账户余额
     */
    public double getBalance() {
        return balance;
    }
}
```

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

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

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

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;


class MyTask implements Callable<Integer> {
    private int upperBounds;

    public MyTask(int upperBounds) {
        this.upperBounds = upperBounds;
    }

    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for(int i = 1; i <= upperBounds; i++) {
            sum += i;
        }
        return sum;
    }

}

class Test {

    public static void main(String[] args) throws Exception {
        List<Future<Integer>> list = new ArrayList<>();
        ExecutorService service = Executors.newFixedThreadPool(10);
```

```
        for(int i = 0; i < 10; i++) {
            list.add(service.submit(new MyTask((int) (Math.random() * 100))));
        }

        int sum = 0;
        for(Future<Integer> future : list) {
            // while(!future.isDone()) ;
            sum += future.get();
        }

        System.out.println(sum);
    }
}
```

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

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

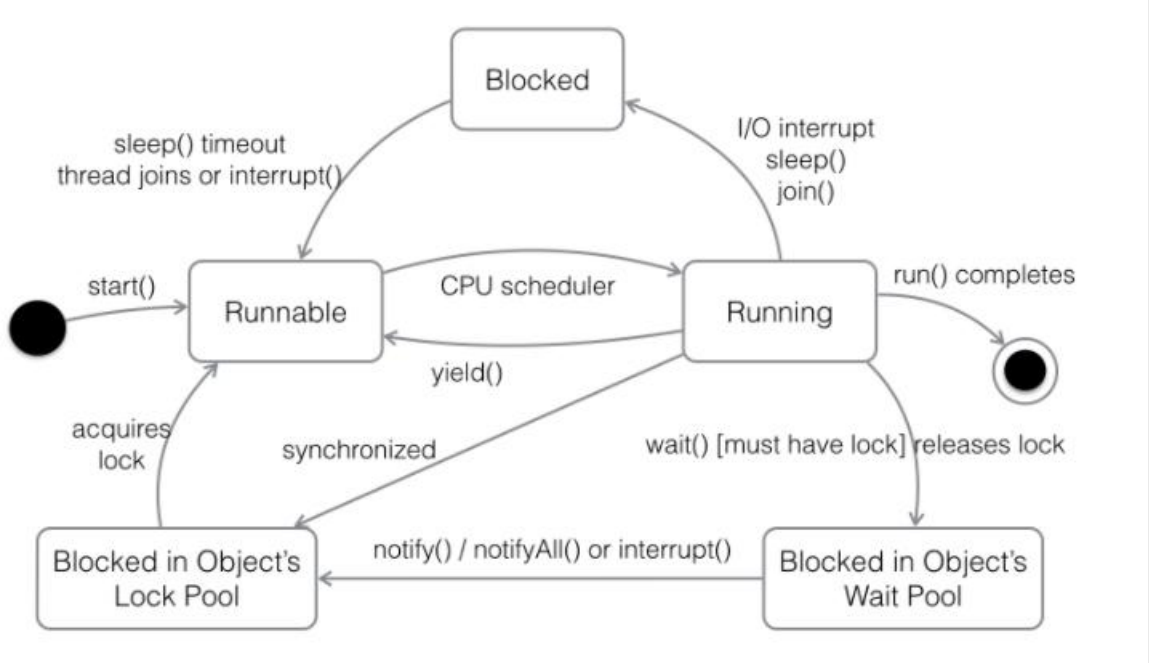
64、启动一个线程是调用 run()还是 start()方法？
答：启动一个线程是调用 start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由 JVM 调度并执行，这并不意味着线程就会立即运行。run()方法是线程启动后要进行回调（callback）的方法。

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

小完全依赖于操作系统（或者说 JVM）能够创建的最大线程大小。
- newScheduledThreadPool：创建一个大小无限的线程池。此线程池支持定时以及周期性执行任务的需求。
- newSingleThreadExecutor：创建一个单线程的线程池。此线程池支持定时以及周期性执行任务的需求。

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

66、线程的基本状态以及状态之间的关系？
答：



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

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

68、Java 中如何实现序列化，有什么意义？
答：序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决对象流读写操作时可能引发的问题（如果不进行序列化可能会存在数据乱序的问题）。要实现序列化，需要让一个类实现 Serializable 接口，该接口是一个标识性接口，标注该类对象是可被序列化的，然后使用一个输出流来构造一个对象输出流并通过

writeObject(Object)方法就可以将实现对象写出（即保存其状态）；如果需要反序列化则可以用一个输入流建立对象输入流，然后通过 readObject 方法从流中读取对象。序列化除了能够实现对象的持久化之外，还能够用于对象的深度克隆（可以参考第 29 题）。

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

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

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;

public final class MyUtil {

    private MyUtil() {
        throw new AssertionError();
    }

    public static void fileCopy(String source, String target) throws IOException {
        try (InputStream in = new FileInputStream(source)) {
            try (OutputStream out = new FileOutputStream(target)) {
                byte[] buffer = new byte[4096];
                int bytesToRead;
                while((bytesToRead = in.read(buffer)) != -1) {
                    out.write(buffer, 0, bytesToRead);
                }
            }
        }
    }

    public static void fileCopyNIO(String source, String target) throws IOException {
        try (FileInputStream in = new FileInputStream(source)) {
            try (FileOutputStream out = new FileOutputStream(target)) {
                FileChannel inChannel = in.getChannel();
                FileChannel outChannel = out.getChannel();
                ByteBuffer buffer = ByteBuffer.allocate(4096);
                while(inChannel.read(buffer) != -1) {
                    buffer.flip();
                    outChannel.write(buffer);
                    buffer.clear();
                }
            }
        }
    }
}
```

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

70、写一个方法，输入一个文件名和一个字符串，统计这个字符串在这个文件中出现的次数。
答：代码如下：

```
import java.io.BufferedReader;
import java.io.FileReader;

public final class MyUtil {

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

    /**
     * 统计给定文件中给定字符串的出现次数
     *
     * @param filename  文件名
     * @param word 字符串
     * @return 字符串在文件中出现的次数
     */
    public static int countWordInFile(String filename, String word) {
        int counter = 0;
        try (FileReader fr = new FileReader(filename)) {
            try (BufferedReader br = new BufferedReader(fr)) {
                String line = null;
                while ((line = br.readLine()) != null) {
                    int index = -1;
                    while (line.length() >= word.length() && (index = line.indexOf(word)) >= 0) {
                        counter++;
                        line = line.substring(index + word.length());
                    }
                }
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return counter;
    }

}
```

71、如何用 Java 代码列出一个目录下所有的文件？
答：
如果只要求列出当前文件夹下的文件，代码如下所示：

```
import java.io.File;

class Test12 {

    public static void main(String[] args) {
        File f = new File("/Users/Hao/Downloads");
        for(File temp : f.listFiles()) {
```

```
            if(temp.isFile()) {
                System.out.println(temp.getName());
            }
        }
    }
}
```

如果需要对文件夹继续展开，代码如下所示：

```
import java.io.File;

class Test12 {

    public static void main(String[] args) {
        showDirectory(new File("/Users/Hao/Downloads"));
    }

    public static void showDirectory(File f) {
        _walkDirectory(f, 0);
    }

    private static void _walkDirectory(File f, int level) {
        if(f.isDirectory()) {
            for(File temp : f.listFiles()) {
                _walkDirectory(temp, level + 1);
            }
        }
        else {
            for(int i = 0; i < level - 1; i++) {
                System.out.print("\t");
            }
            System.out.println(f.getName());
        }
    }
}
```

在 Java 7 中可以使用 NIO.2 的 API 来做同样的事情，代码如下所示：

```
class ShowFileTest {

    public static void main(String[] args) throws IOException {
        Path initPath = Paths.get("/Users/Hao/Downloads");
        Files.walkFileTree(initPath, new SimpleFileVisitor<Path>() {

            @Override
            public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
                    throws IOException {
                System.out.println(file.getFileName().toString());
                return FileVisitResult.CONTINUE;
            }

        });
    }
}
```

72、用 Java 的套接字编程实现一个多线程的回显（echo）服务器。
答：

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoServer {

   private static final int ECHO_SERVER_PORT = 6789;

   public static void main(String[] args) {
      try(ServerSocket server = new ServerSocket(ECHO_SERVER_PORT)) {
         System.out.println("服务器已经启动...");
         while(true) {
            Socket client = server.accept();
            new Thread(new ClientHandler(client)).start();
         }
      } catch (IOException e) {
         e.printStackTrace();
      }
   }

   private static class ClientHandler implements Runnable {
      private Socket client;

      public ClientHandler(Socket client) {
         this.client = client;
      }

      @Override
      public void run() {
         try(BufferedReader br = new BufferedReader(new
InputStreamReader(client.getInputStream()));
               PrintWriter pw = new PrintWriter(client.getOutputStream())) {
            String msg = br.readLine();
            System.out.println("收到" + client.getInetAddress() + "发送的: " + msg);
            pw.println(msg);
            pw.flush();
         } catch(Exception ex) {
            ex.printStackTrace();
         } finally {
            try {
               client.close();
            } catch (IOException e) {
               e.printStackTrace();
            }
         }
      }
   }

}
```

注意：上面的代码使用了 Java 7 的 TWR 语法，由于很多外部资源类都间接的实现了 AutoCloseable 接口（单方法回调接口），因此可以利用 TWR 语法在 try 结束的时候通过回调的方式自动调用外部资源类的 close()方法，避免书写冗长的 finally 代码块。此外，上面

的代码用一个静态内部类实现线程的功能，使用多线程可以避免一个用户 I/O 操作所产生的中断影响其他用户对服务器的访问，简单的说就是一个用户的输入操作不会造成其他用户的阻塞。当然，上面的代码使用线程池可以获得更好的性能，因为频繁的创建和销毁线程所造成的开销也是不可忽视的。

下面是一段回显客户端测试代码：

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.Scanner;

public class EchoClient {

    public static void main(String[] args) throws Exception {
        Socket client = new Socket("localhost", 6789);
        Scanner sc = new Scanner(System.in);
        System.out.print("请输入内容: ");
        String msg = sc.nextLine();
        sc.close();
        PrintWriter pw = new PrintWriter(client.getOutputStream());
        pw.println(msg);
        pw.flush();
        BufferedReader br = new BufferedReader(new InputStreamReader(client.getInputStream()));
        System.out.println(br.readLine());
        client.close();
    }
}
```

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

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.Iterator;

public class EchoServerNIO {

    private static final int ECHO_SERVER_PORT = 6789;
    private static final int ECHO_SERVER_TIMEOUT = 5000;
    private static final int BUFFER_SIZE = 1024;

    private static ServerSocketChannel serverChannel = null;
    private static Selector selector = null;    // 多路复用选择器
    private static ByteBuffer buffer = null;    // 缓冲区

    public static void main(String[] args) {
        init();
        listen();
```

```
}

private static void init() {
    try {
        serverChannel = ServerSocketChannel.open();
        buffer = ByteBuffer.allocate(BUFFER_SIZE);
        serverChannel.socket().bind(new InetSocketAddress(ECHO_SERVER_PORT));
        serverChannel.configureBlocking(false);
        selector = Selector.open();
        serverChannel.register(selector, SelectionKey.OP_ACCEPT);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}

private static void listen() {
    while (true) {
        try {
            if (selector.select(ECHO_SERVER_TIMEOUT) != 0) {
                Iterator<SelectionKey> it = selector.selectedKeys().iterator();
                while (it.hasNext()) {
                    SelectionKey key = it.next();
                    it.remove();
                    handleKey(key);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

private static void handleKey(SelectionKey key) throws IOException {
    SocketChannel channel = null;

    try {
        if (key.isAcceptable()) {
            ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
            channel = serverChannel.accept();
            channel.configureBlocking(false);
            channel.register(selector, SelectionKey.OP_READ);
        } else if (key.isReadable()) {
            channel = (SocketChannel) key.channel();
            buffer.clear();
            if (channel.read(buffer) > 0) {
                buffer.flip();
                CharBuffer charBuffer = CharsetHelper.decode(buffer);
                String msg = charBuffer.toString();
                System.out.println("收到" + channel.getRemoteAddress() + "的消息：" + msg);
                channel.write(CharsetHelper.encode(CharBuffer.wrap(msg)));
            } else {
                channel.close();
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
        if (channel != null) {
```

```
            channel.close();
        }
    }
  }

}
```

```
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.charset.CharacterCodingException;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import java.nio.charset.CharsetEncoder;

public final class CharsetHelper {
   private static final String UTF_8 = "UTF-8";
   private static CharsetEncoder encoder = Charset.forName(UTF_8).newEncoder();
   private static CharsetDecoder decoder = Charset.forName(UTF_8).newDecoder();

   private CharsetHelper() {
   }

   public static ByteBuffer encode(CharBuffer in) throws CharacterCodingException{
      return encoder.encode(in);
   }

   public static CharBuffer decode(ByteBuffer in) throws CharacterCodingException{
      return decoder.decode(in);
   }
}
```

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

74、你在项目中哪些地方用到了 XML？
答：XML 的主要作用有两个方面：数据交换和信息配置。在做数据交换时，XML 将数据用标签组装成起来，然后压缩打包加密后通过网络传送给接收者，接收解密与解压缩后再从 XML 文件中还原相关信息进行处理，XML 曾经是异构系统间交换数据的事实标准，但此项功能几乎已经被 JSON（JavaScriptObjectNotation）取而代之。当然，目前很多软件仍然使用 XML 来存储配置信息，我们在很多项目中通常也会将作为配置信息的硬代码写在

XML 文件中，Java 的很多框架也是这么做的，而且这些框架都选择了 dom4j 作为处理 XML 的工具，因为 Sun 公司的官方 API 实在不怎么好用。

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

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

加载驱动。

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

创建连接。

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

创建语句。

```
PreparedStatement ps = con.prepareStatement("select * from emp where sal between ? and ?");
ps.setInt(1, 1000);
ps.setInt(2, 3000);
```

执行语句。

```
ResultSet rs = ps.executeQuery();
```

处理结果。

```
while(rs.next()) {
    System.out.println(rs.getInt("empno") + " - " + rs.getString("ename"));
}
```

关闭资源。

```
finally {
    if(con != null) {
        try {
            con.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

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

76、Statement 和 PreparedStatement 有什么区别？哪个性能更好？
答：与 Statement 相比，①PreparedStatement 接口代表预编译的语句，它主要的优势在于可以减少 SQL 的编译错误并增加 SQL 的安全性（减少 SQL 注射攻击的可能性）；②

PreparedStatement 中的 SQL 语句是可以带参数的，避免了用字符串连接拼接 SQL 语句的麻烦和不安全；③当批量处理 SQL 或频繁执行相同的查询时，PreparedStatement 有明显的性能上的优势，由于数据库可以将编译优化后的 SQL 语句缓存起来，下次执行相同结构的语句时就会很快（不用再次编译和生成执行计划）。

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

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

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

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

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

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

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

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

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

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

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

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

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

第 1 类丢失更新：事务 A 撤销时，把已经提交的事务 B 的更新数据覆盖了。

时间	取款事务 A	转账事务 B

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

第 2 类丢失更新：事务 A 覆盖事务 B 已经提交的数据，造成事务 B 所做的操作丢失。

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

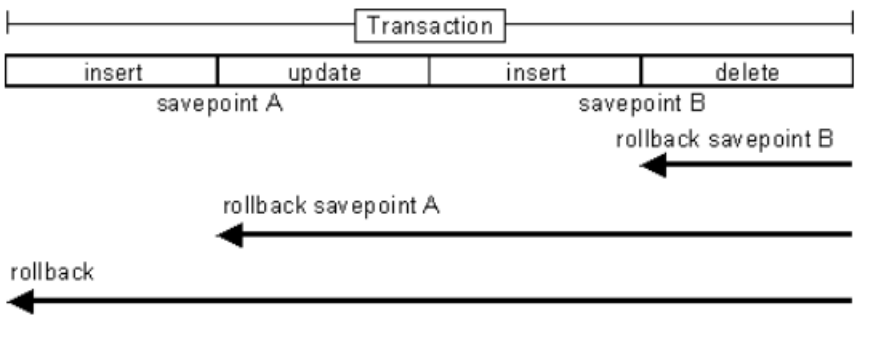
数据并发访问所产生的问题，在有些场景下可能是允许的，但是有些场景下可能就是致命的，数据库通常会通过锁机制来解决数据并发访问问题，按锁定对象不同可以分为表级锁和行级锁；按并发事务锁定关系可以分为共享锁和独占锁，具体的内容大家可以自行查阅资料进行了解。直接使用锁是非常麻烦的，为此数据库为用户提供了自动锁机制，只要用户指定会话的事务隔离级别，数据库就会通过分析 SQL 语句然后为事务访问的资源加上合适的锁，此外，数据库还会维护这些锁通过各种手段提高系统的性能，这些对用户来说都是透明的（就是说你不用理解，事实上我确实也不知道）。ANSI/ISO SQL 92 标准定义了 4 个等级的事务隔离级别，如下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

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

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

81、JDBC 中如何进行事务处理？答：Connection 提供了事务处理的方法，通过调用 setAutoCommit(false)可以设置手动提交事务；当事务完成后用 commit()显式提交事务；如果在事务处理过程中发生异常则通过 rollback()进行事务回滚。除此之外，从 JDBC 3.0 中

还引入了 Savepoint（保存点）的概念，允许通过代码设置保存点并让事务回滚到指定的保存点。



82、JDBC 能否处理 Blob 和 Clob?

答： Blob 是指二进制大对象（Binary Large Object），而 Clob 是指大字符对象（Character Large Objec），因此其中 Blob 是为存储大的二进制数据而设计的，而 Clob 是为存储大的文本数据而设计的。JDBC 的 PreparedStatement 和 ResultSet 都提供了相应的方法来支持 Blob 和 Clob 操作。下面的代码展示了如何使用 JDBC 操作 LOB：

下面以 MySQL 数据库为例，创建一个张有三个字段的用户表，包括编号（id）、姓名（name）和照片（photo），建表语句如下：

```
create table tb_user
(
id int primary key auto_increment,
name varchar(20) unique not null,
photo longblob
);
```

下面的 Java 代码向数据库中插入一条记录：

```
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

class JdbcLobTest {

    public static void main(String[] args) {
        Connection con = null;
        try {
            // 1. 加载驱动（Java6 以上版本可以省略）
            Class.forName("com.mysql.jdbc.Driver");
            // 2. 建立连接
            con = DriverManager.getConnection("jdbc:mysql://localhost:3306/test", "root", "123456");
            // 3. 创建语句对象
            PreparedStatement ps = con.prepareStatement("insert into tb_user values (default, ?, ?)");
            ps.setString(1, "骆昊");            // 将 SQL 语句中第一个占位符换成字符串
            try (InputStream in = new FileInputStream("test.jpg")) {    // Java 7 的 TWR
                ps.setBinaryStream(2, in);      // 将 SQL 语句中第二个占位符换成二进制流
                // 4. 发出 SQL 语句获得受影响行数
                System.out.println(ps.executeUpdate() == 1 ? "插入成功" : "插入失败");
```

```
            } catch(IOException e) {
                System.out.println("读取照片失败!");
            }
        } catch (ClassNotFoundException | SQLException e) {    // Java 7 的多异常捕获
            e.printStackTrace();
        } finally { // 释放外部资源的代码都应当放在 finally 中保证其能够得到执行
            try {
                if(con != null && !con.isClosed()) {
                    con.close();    // 5. 释放数据库连接
                    con = null;    // 指示垃圾回收器可以回收该对象
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

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

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

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

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

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class RegExpTest {

    public static void main(String[] args) {
        String str = "北京市(朝阳区)(西城区)(海淀区)";
        Pattern p = Pattern.compile(".*?(?=\\()");
        Matcher m = p.matcher(str);
        if(m.find()) {
            System.out.println(m.group());
        }
    }
}
```

说明：上面的正则表达式中使用了懒惰匹配和前瞻，如果不清楚这些内容，推荐读一下网上很有名的《正则表达式 30 分钟入门教程》。

85、获得一个类的类对象有哪些方式？
答：

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

86、如何通过反射创建实例对象？
答：
- 方法 1：通过类对象调用 newInstance()方法，例如：String.class.newInstance()
- 方法 2：通过类对象的 getConstructor()或 getDeclaredConstructor()方法获得构造器（Constructor）对象并调用其 newInstance()方法创建对象，例如：String.class.getConstructor(String.class).newInstance("Hello");

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

```
import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Modifier;
import java.util.ArrayList;
import java.util.List;

/**
 * 反射工具类
 * @author 骆昊
 *
 */
public class ReflectionUtil {

    private ReflectionUtil() {
        throw new AssertionError();
    }

    /**
     * 通过反射取对象指定字段(属性)的值
     * @param target 目标对象
     * @param fieldName 字段的名字
     * @throws 如果取不到对象指定字段的值则抛出异常
     * @return 字段的值
     */
    public static Object getValue(Object target, String fieldName) {
        Class<?> clazz = target.getClass();
        String[] fs = fieldName.split("\\.");

        try {
            for(int i = 0; i < fs.length - 1; i++) {
                Field f = clazz.getDeclaredField(fs[i]);
                f.setAccessible(true);
                target = f.get(target);
                clazz = target.getClass();
            }

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

```
            f.setAccessible(true);
            return f.get(target);
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    /**
     * 通过反射给对象的指定字段赋值
     * @param target 目标对象
     * @param fieldName 字段的名称
     * @param value 值
     */
    public static void setValue(Object target, String fieldName, Object value) {
        Class<?> clazz = target.getClass();
        String[] fs = fieldName.split("\\.");
        try {
            for(int i = 0; i < fs.length - 1; i++) {
                Field f = clazz.getDeclaredField(fs[i]);
                f.setAccessible(true);
                Object val = f.get(target);
                if(val == null) {
                    Constructor<?> c = f.getType().getDeclaredConstructor();
                    c.setAccessible(true);
                    val = c.newInstance();
                    f.set(target, val);
                }
                target = val;
                clazz = target.getClass();
            }

            Field f = clazz.getDeclaredField(fs[fs.length - 1]);
            f.setAccessible(true);
            f.set(target, value);
        }
        catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

88、如何通过反射调用对象的方法？
答：请看下面的代码：

```
import java.lang.reflect.Method;

class MethodInvokeTest {

    public static void main(String[] args) throws Exception {
        String str = "hello";
        Method m = str.getClass().getMethod("toUpperCase");
        System.out.println(m.invoke(str));  // HELLO
    }
}
```

89、简述一下面向对象的"六原则一法则"。

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

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

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

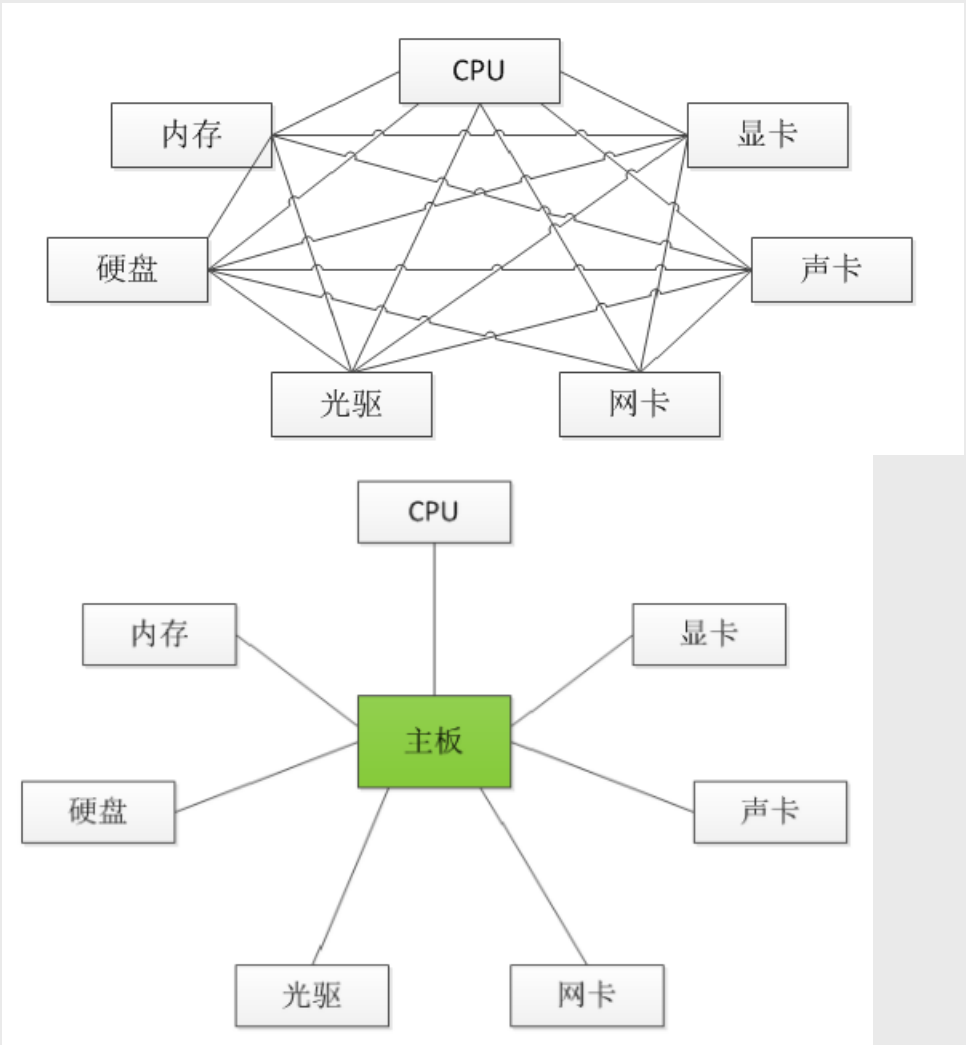
-里氏替换原则：任何时候都可以用子类型替换掉父类型。（关于里氏替换原则的描述，Barbara Liskov 女士的描述比这个要复杂得多，但简单的说就是能用父类型的地方就一定能使用子类型。里氏替换原则可以检查继承关系是否合理，如果一个继承关系违背了里氏替换原则，那么这个继承关系一定是错误的，需要对代码进行重构。例如让猫继承狗，或者狗继承猫，又或者让正方形继承长方形都是错误的继承关系，因为你很容易找到违反里氏替换原则的场景。需要注意的是：子类一定是增加父类的能力而不是减少父类的能力，因为子类比父类的能力更多，把能力多的对象当成能力少的对象来用当然没有任何问题。）

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

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

- 迪米特法则：迪米特法则又叫最少知识原则，一个对象应当对其他对象有尽可能少的了解。（迪米特法则简单的说就是如何做到"低耦合"，门面模式和调停者模式就是对迪米特法则的践行。对于门面模式可以举一个简单的例子，你去一家公司洽谈业务，你不需要了解这个公司内部是如何运作的，你甚至可以对这个公司一无所知，去的时候只需要找到公司入口处的前台美女，告诉她们你要做什么，她们会找到合适的人跟你接洽，前台的美女

就是公司这个系统的门面。再复杂的系统都可以为用户提供一个简单的门面，Java Web 开发中作为前端控制器的 Servlet 或 Filter 不就是一个门面吗，浏览器对服务器的运作方式一无所知，但是通过前端控制器就能够根据你的请求得到相应的服务。调停者模式也可以举一个简单的例子来说明，例如一台计算机，CPU、内存、硬盘、显卡、声卡各种设备需要相互配合才能很好的工作，但是如果这些东西都直接连接到一起，计算机的布线将异常复杂，在这种情况下，主板作为一个调停者的身份出现，它将各个设备连接在一起而不需要每个设备之间直接交换数据，这样就减小了系统的耦合度和复杂度，如下图所示。迪米特法则用通俗的话来将就是不要和陌生人打交道，如果真的需要，找一个自己的朋友，让他替你和陌生人打交道。）



90、简述一下你了解的设计模式。

答：所谓设计模式，就是一套被反复使用的代码设计经验的总结（情境中一个问题经过证实的一个解决方案）。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式使人们可以更加简单方便的复用成功的设计和体系结构。将已证实的技术表述成设计模式也会使新系统开发者更加容易理解其设计思路。

在 GoF 的《Design Patterns: Elements of Reusable Object-Oriented Software》中给出了三类（创建型[对类的实例化过程的抽象化]、结构型[描述如何将类或对象结合在一起形成更大的结构]、行为型[对在不同的对象之间划分责任和算法的抽象化]）共 23 种设计模式，包括：Abstract Factory（抽象工厂模式），Builder（建造者模式），Factory Method（工厂方法模式），Prototype（原始模型模式），Singleton（单例模式）；Facade（门面模式），Adapter（适配器模式），Bridge（桥梁模式），Composite（合成模式），Decorator（装饰模式），Flyweight（享元模式），Proxy（代理模式）；Command（命令模式），

Interpreter（解释器模式），Visitor（访问者模式），Iterator（迭代子模式），Mediator（调停者模式），Memento（备忘录模式），Observer（观察者模式），State（状态模式），Strategy（策略模式），Template Method（模板方法模式），Chain Of Responsibility（责任链模式）。
面试被问到关于设计模式的知识时，可以拣最常用的作答，例如：
- 工厂模式：工厂类可以根据条件生成不同的子类实例，这些子类有一个公共的抽象父类并且实现了相同的方法，但是这些方法针对不同的数据进行了不同的操作（多态方法）。当得到子类的实例后，开发人员可以调用基类中的方法而不必考虑到底返回的是哪一个子类的实例。
- 代理模式：给一个对象提供一个代理对象，并由代理对象控制原对象的引用。实际开发中，按照使用目的的不同，代理可以分为：远程代理、虚拟代理、保护代理、Cache 代理、防火墙代理、同步化代理、智能引用代理。
- 适配器模式：把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起使用的类能够一起工作。
- 模板方法模式：提供一个抽象类，将部分逻辑以具体方法或构造器的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法（多态实现），从而实现不同的业务逻辑。
除此之外，还可以讲讲上面提到的门面模式、桥梁模式、单例模式、装潢模式（Collections 工具类和 I/O 系统中都使用装潢模式）等，反正基本原则就是拣自己最熟悉的、用得最多的作答，以免言多必失。

91、用 Java 写一个单例类。
答：
- 饿汉式单例

```
public class Singleton {
   private Singleton(){}
   private static Singleton instance = new Singleton();
   public static Singleton getInstance(){
      return instance;
   }
}
```

懒汉式单例

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

双重检测锁模式单例

```
class Singleton4{
        private Singleton4(){
        }
        private static Singleton4 single = null;
        public static Singleton4 getInstance(){
                if(single==null){
                        synchronized(Singleton4.class){
                                if(single==null){
                                        single=new Singleton4();
                                }
```

```
                    }
                }
                return single;
            }
}
```

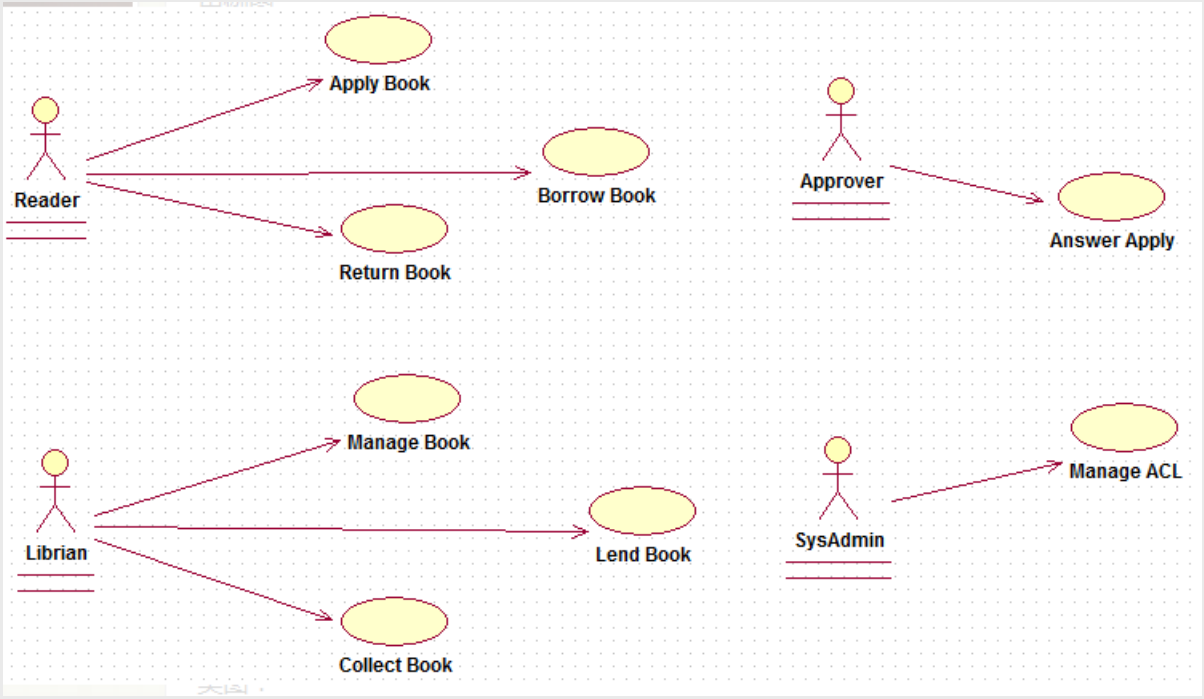
注意：实现一个单例有两点注意事项，①将构造器私有，不允许外界通过构造器创建对象；②通过公开的静态方法向外界返回类的唯一实例。这里有一个问题可以思考：Spring 的 IoC 容器可以为普通的类创建单例，它是怎么做到的呢？

92、什么是 UML？

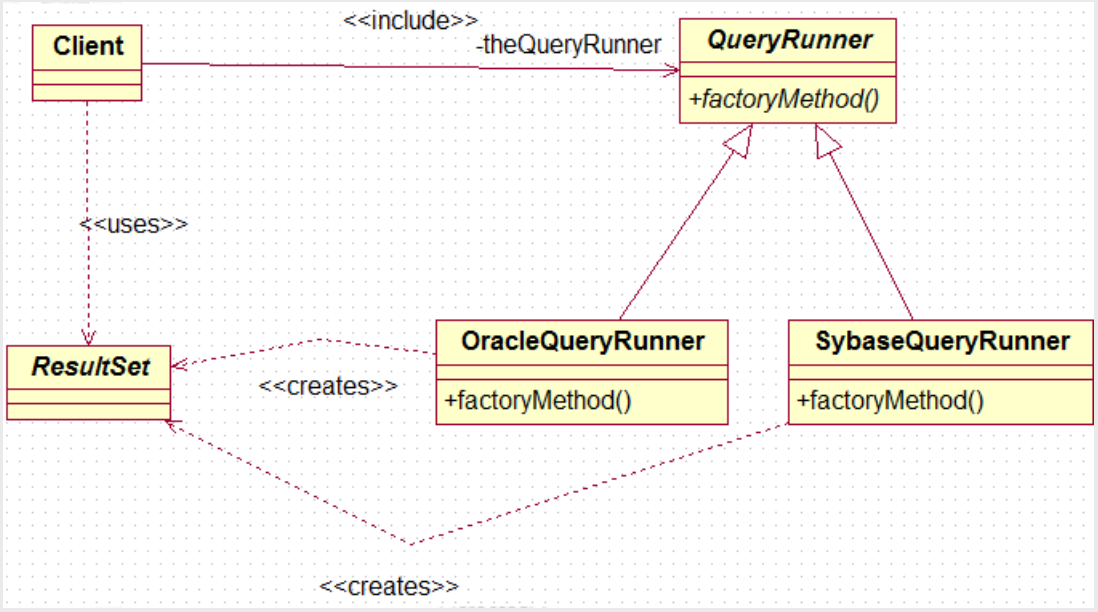
答：UML 是统一建模语言（Unified Modeling Language）的缩写，它发表于 1997 年，综合了当时已经存在的面向对象的建模语言、方法和过程，是一个支持模型化和软件系统开发的图形化语言，为软件开发的所有阶段提供模型化和可视化支持。使用 UML 可以帮助沟通与交流，辅助应用设计和文档的生成，还能够阐释系统的结构和行为。

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

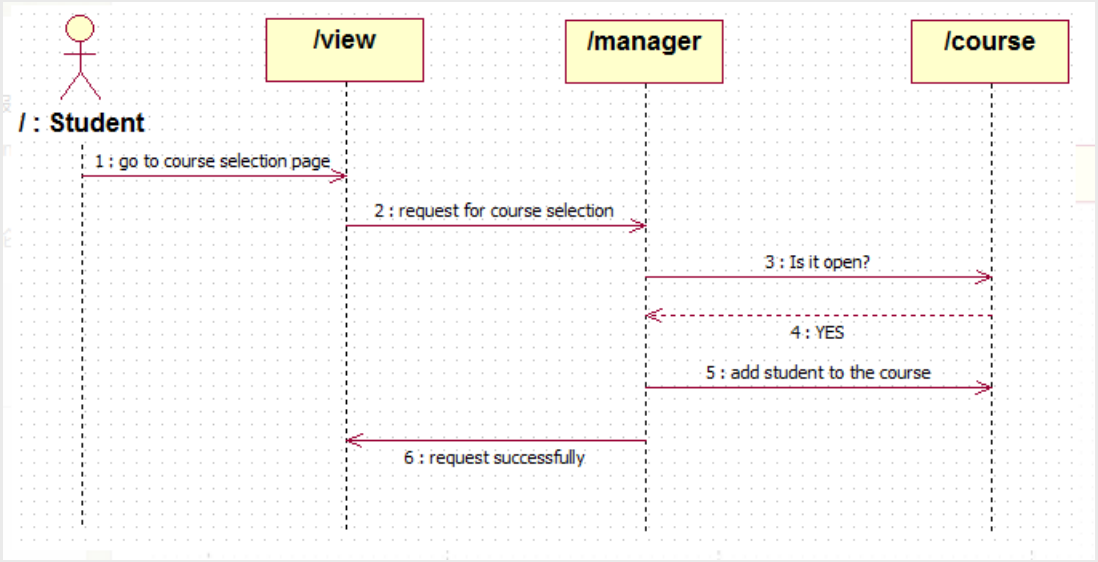
用例图：



类图：



时序图：



94、用 Java 写一个冒泡排序。

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

```
import java.util.Comparator;

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

  /**
   * 排序
   * @param list 待排序的数组
   */
  public <T extends Comparable<T>> void sort(T[] list);
```

```
  /**
   * 排序
   * @param list 待排序的数组
   * @param comp 比较两个对象的比较器
   */
  public <T> void sort(T[] list, Comparator<T> comp);
}

import java.util.Comparator;

/**
 * 冒泡排序
 *
 * @author 骆昊
 *
 */
public class BubbleSorter implements Sorter {

    @Override
    public <T extends Comparable<T>> void sort(T[] list) {
        boolean swapped = true;
        for (int i = 1, len = list.length; i < len && swapped; ++i) {
            swapped = false;
            for (int j = 0; j < len - i; ++j) {
                if (list[j].compareTo(list[j + 1]) > 0) {
                    T temp = list[j];
                    list[j] = list[j + 1];
                    list[j + 1] = temp;
                    swapped = true;
                }
            }
        }
    }

    @Override
    public <T> void sort(T[] list, Comparator<T> comp) {
        boolean swapped = true;
        for (int i = 1, len = list.length; i < len && swapped; ++i) {
            swapped = false;
            for (int j = 0; j < len - i; ++j) {
                if (comp.compare(list[j], list[j + 1]) > 0) {
                    T temp = list[j];
                    list[j] = list[j + 1];
                    list[j + 1] = temp;
                    swapped = true;
                }
            }
        }
    }
}
```

95、用 Java 写一个折半查找。
答：折半查找，也称二分查找、二分搜索，是一种在有序数组中查找某一特定元素的搜索算法。搜素过程从数组的中间元素开始，如果中间元素正好是要查找的元素，则搜素过程结束；如果某一特定元素大于或者小于中间元素，则在数组大于或小于中间元素的那一半

中查找，而且跟开始一样从中间元素开始比较。如果在某一步骤数组已经为空，则表示找不到指定的元素。这种搜索算法每一次比较都使搜索范围缩小一半，其时间复杂度是O(logN)。

```
import java.util.Comparator;

public class MyUtil {

  public static <T extends Comparable<T>> int binarySearch(T[] x, T key) {
    return binarySearch(x, 0, x.length- 1, key);
  }

  // 使用循环实现的二分查找
  public static <T> int binarySearch(T[] x, T key, Comparator<T> comp) {
    int low = 0;
    int high = x.length - 1;
    while (low <= high) {
      int mid = (low + high) >>> 1;
      int cmp = comp.compare(x[mid], key);
      if (cmp < 0) {
        low= mid + 1;
      }
      else if (cmp > 0) {
        high= mid - 1;
      }
      else {
        return mid;
      }
    }
    return -1;
  }

  // 使用递归实现的二分查找
  private static<T extends Comparable<T>> int binarySearch(T[] x, int low, int high, T key) {
    if(low <= high) {
      int mid = low + ((high -low) >> 1);
      if(key.compareTo(x[mid])== 0) {
        return mid;
      }
      else if(key.compareTo(x[mid])< 0) {
        return binarySearch(x,low, mid - 1, key);
      }
      else {
        return binarySearch(x,mid + 1, high, key);
      }
    }
    return -1;
  }
}
```

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

这部分主要是与 Java Web 和 Web Service 相关的面试题。

96、阐述 Servlet 和 CGI 的区别?
答：Servlet 与 CGI 的区别在于 Servlet 处于服务器进程中，它通过多线程方式运行其 service()方法，一个实例可以服务于多个请求，并且其实例一般不会销毁，而 CGI 对每个请求都产生新的进程，服务完成后就销毁，所以效率上低于 Servlet。

补充：Sun Microsystems 公司在 1996 年发布 Servlet 技术就是为了和 CGI 进行竞争，Servlet 是一个特殊的 Java 程序，一个基于 Java 的 Web 应用通常包含一个或多个 Servlet 类。Servlet 不能够自行创建并执行，它是在 Servlet 容器中运行的，容器将用户的请求传递给 Servlet 程序，并将 Servlet 的响应回传给用户。通常一个 Servlet 会关联一个或多个 JSP 页面。以前 CGI 经常因为性能开销上的问题被诟病，然而 Fast CGI 早就已经解决了 CGI 效率上的问题，所以面试的时候大可不必信口开河的诟病 CGI，事实上有很多你熟悉的网站都使用了 CGI 技术。

97、Servlet 接口中有哪些方法？
答：Servlet 接口定义了 5 个方法，其中前三个方法与 Servlet 生命周期相关：
- void init(ServletConfig config) throws ServletException
- void service(ServletRequest req, ServletResponse resp) throws ServletException, java.io.IOException
- void destory()
- java.lang.String getServletInfo()
- ServletConfig getServletConfig()

Web 容器加载 Servlet 并将其实例化后，Servlet 生命周期开始，容器运行其 init()方法进行 Servlet 的初始化；请求到达时调用 Servlet 的 service()方法，service()方法会根据需要调用与请求对应的 doGet 或 doPost 等方法；当服务器关闭或项目被卸载时服务器会将 Servlet 实例销毁，此时会调用 Servlet 的 destroy()方法。

98、转发（forward）和重定向（redirect）的区别？
答：forward 是容器中控制权的转向，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，然后把这些内容再发给浏览器，浏览器根本不知道服务器发送的内容是从哪儿来的，所以它的地址栏中还是原来的地址。redirect 就是服务器端根据逻辑，发送一个状态码，告诉浏览器重新去请求那个地址，因此从浏览器的地址栏中可以看到跳转后的链接地址，很明显 redirect 无法访问到服务器保护起来资源，但是可以从一个网站 redirect 到其他网站。forward 更加高效，所以在满足需要时尽量使用 forward（通过调用 RequestDispatcher 对象的 forward()方法，该对象可以通过 ServletRequest 对象的 getRequestDispatcher()方法获得），并且这样也有助于隐藏实际的链接；在有些情况下，比如需要访问一个其它服务器上的资源，则必须使用重定向（通过 HttpServletResponse 对象调用其 sendRedirect()方法实现）。

99、JSP 有哪些内置对象？作用分别是什么？
答：JSP 有 9 个内置对象：
- request：封装客户端的请求，其中包含来自 GET 或 POST 请求的参数；
- response：封装服务器对客户端的响应；
- pageContext：通过该对象可以获取其他对象；
- session：封装用户会话的对象；
- application：封装服务器运行环境的对象；
- out：输出服务器响应的输出流对象；
- config：Web 应用的配置对象；
- page：JSP 页面本身（相当于 Java 程序中的 this）；
- exception：封装页面抛出异常的对象。

补充：如果用 Servlet 来生成网页中的动态内容无疑是非常繁琐的工作，另一方面，所有

的文本和 HTML 标签都是硬编码，即使做出微小的修改，都需要进行重新编译。JSP 解决了 Servlet 的这些问题，它是 Servlet 很好的补充，可以专门用作为用户呈现视图（View），而 Servlet 作为控制器（Controller）专门负责处理用户请求并转发或重定向到某个页面。基于 Java 的 Web 开发很多都同时使用了 Servlet 和 JSP。JSP 页面其实是一个 Servlet，能够运行 Servlet 的服务器（Servlet 容器）通常也是 JSP 容器，可以提供 JSP 页面的运行环境，Tomcat 就是一个 Servlet/JSP 容器。第一次请求一个 JSP 页面时，Servlet/JSP 容器首先将 JSP 页面转换成一个 JSP 页面的实现类，这是一个实现了 JspPage 接口或其子接口 HttpJspPage 的 Java 类。JspPage 接口是 Servlet 的子接口，因此每个 JSP 页面都是一个 Servlet。转换成功后，容器会编译 Servlet 类，之后容器加载和实例化 Java 字节码，并执行它通常对 Servlet 所做的生命周期操作。对同一个 JSP 页面的后续请求，容器会查看这个 JSP 页面是否被修改过，如果修改过就会重新转换并重新编译并执行。如果没有则执行内存中已经存在的 Servlet 实例。我们可以看一段 JSP 代码对应的 Java 程序就知道一切了，而且 9 个内置对象的神秘面纱也会被揭开。

JSP 页面：

```
<%@ page pageEncoding="UTF-8"%>
<%
String path = request.getContextPath();
String basePath = request.getScheme() + "://" + request.getServerName() + ":" +
request.getServerPort() + path + "/";
%>

<!DOCTYPE html>
<html>
  <head>
    <base href="<%=basePath%>">
    <title>首页</title>
    <style type="text/css">
        * { font-family: "Arial"; }
    </style>
  </head>

  <body>
    <h1>Hello, World!</h1>
    <hr/>
    <h2>Current time is: <%= new java.util.Date().toString() %></h2>
  </body>
</html>
```

对应的 Java 代码：

```
package org.apache.jsp;

import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class index_jsp extends org.apache.jasper.runtime.HttpJspBase
      implements org.apache.jasper.runtime.JspSourceDependent {

   private static final javax.servlet.jsp.JspFactory _jspxFactory = javax.servlet.jsp.JspFactory
         .getDefaultFactory();

   private static java.util.Map<java.lang.String, java.lang.Long> _jspx_dependants;
```

```
  private javax.el.ExpressionFactory _el_expressionfactory;
  private org.apache.tomcat.InstanceManager _jsp_instancemanager;

  public java.util.Map<java.lang.String, java.lang.Long> getDependants() {
    return _jspx_dependants;
  }

  public void _jspInit() {
    _el_expressionfactory = _jspxFactory.getJspApplicationContext(
        getServletConfig().getServletContext()).getExpressionFactory();
    _jsp_instancemanager = org.apache.jasper.runtime.InstanceManagerFactory
        .getInstanceManager(getServletConfig());
  }

  public void _jspDestroy() {
  }

  public void _jspService(
      final javax.servlet.http.HttpServletRequest request,
      final javax.servlet.http.HttpServletResponse response)
      throws java.io.IOException, javax.servlet.ServletException {
    // 内置对象就是在这里定义的
    final javax.servlet.jsp.PageContext pageContext;
    javax.servlet.http.HttpSession session = null;
    final javax.servlet.ServletContext application;
    final javax.servlet.ServletConfig config;
    javax.servlet.jsp.JspWriter out = null;
    final java.lang.Object page = this;
    javax.servlet.jsp.JspWriter _jspx_out = null;
    javax.servlet.jsp.PageContext _jspx_page_context = null;

    try {
      response.setContentType("text/html;charset=UTF-8");
      pageContext = _jspxFactory.getPageContext(this, request, response,
          null, true, 8192, true);
      _jspx_page_context = pageContext;
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;

      out.write('\r');
      out.write('\n');

      String path = request.getContextPath();
      String basePath = request.getScheme() + "://"
          + request.getServerName() + ":" + request.getServerPort()
          + path + "/";
// 以下代码通过输出流将 HTML 标签输出到浏览器中
      out.write("\r\n");
      out.write("\r\n");
      out.write("<!DOCTYPE html>\r\n");
      out.write("<html>\r\n");
      out.write("  <head>\r\n");
      out.write("    <base href=\"");
```

```
            out.print(basePath);
            out.write("\">\r\n");
            out.write("    <title>首页</title>\r\n");
            out.write("    <style type=\"text/css\">\r\n");
            out.write("    \t* { font-family: \"Arial\"; }\r\n");
            out.write("    </style>\r\n");
            out.write("  </head>\r\n");
            out.write("  \r\n");
            out.write("  <body>\r\n");
            out.write("    <h1>Hello, World!</h1>\r\n");
            out.write("    <hr/>\r\n");
            out.write("    <h2>Current time is: ");
            out.print(new java.util.Date().toString());
            out.write("</h2>\r\n");
            out.write("  </body>\r\n");
            out.write("</html>\r\n");
        } catch (java.lang.Throwable t) {
            if (!(t instanceof javax.servlet.jsp.SkipPageException)) {
                out = _jspx_out;
                if (out != null && out.getBufferSize() != 0)
                    try {
                        out.clearBuffer();
                    } catch (java.io.IOException e) {
                    }
                if (_jspx_page_context != null)
                    _jspx_page_context.handlePageException(t);
                else
                    throw new ServletException(t);
            }
        } finally {
            _jspxFactory.releasePageContext(_jspx_page_context);
        }
    }
}
```

100、get 和 post 请求的区别？

答：

①get 请求用来从服务器上获得资源，而 post 是用来向服务器提交数据；

②get 将表单中数据按照 name=value 的形式，添加到 action 所指向的 URL 后面，并且两者使用"?"连接，而各个变量之间使用"&"连接；post 是将表单中的数据放在 HTTP 协议的请求头或消息体中，传递到 action 所指向 URL；

③get 传输的数据要受到 URL 长度限制（1024 字节）；而 post 可以传输大量的数据，上传文件通常要使用 post 方式；

④使用 get 时参数会显示在地址栏上，如果这些数据不是敏感数据，那么可以使用 get；对于敏感数据还是应用使用 post；

⑤get 使用 MIME 类型 application/x-www-form-urlencoded 的 URL 编码（也叫百分号编码）文本的格式传递参数，保证被传送的参数由遵循规范的文本组成，例如一个空格的编码是"%20"。

101、常用的 Web 服务器有哪些？

答：Unix 和 Linux 平台下使用最广泛的免费 HTTP 服务器是 Apache 服务器，而 Windows 平台的服务器通常使用 IIS 作为 Web 服务器。选择 Web 服务器应考虑的因素有：性能、安全性、日志和统计、虚拟主机、代理服务器、缓冲服务和集成应用程序等。下面是对常见服务器的简介：

- IIS：Microsoft 的 Web 服务器产品，全称是 Internet Information Services。IIS 是允许在公

共 Intranet 或 Internet 上发布信息的 Web 服务器。IIS 是目前最流行的 Web 服务器产品之一，很多著名的网站都是建立在 IIS 的平台上。IIS 提供了一个图形界面的管理工具，称为 Internet 服务管理器，可用于监视配置和控制 Internet 服务。IIS 是一种 Web 服务组件，其中包括 Web 服务器、FTP 服务器、NNTP 服务器和 SMTP 服务器，分别用于网页浏览、文件传输、新闻服务和邮件发送等方面，它使得在网络（包括互联网和局域网）上发布信息成了一件很容易的事。它提供 ISAPI(Intranet Server API）作为扩展 Web 服务器功能的编程接口；同时，它还提供一个 Internet 数据库连接器，可以实现对数据库的查询和更新。
- Kangle：Kangle Web 服务器是一款跨平台、功能强大、安全稳定、易操作的高性能 Web 服务器和反向代理服务器软件。此外，Kangle 也是一款专为做虚拟主机研发的 Web 服务器。实现虚拟主机独立进程、独立身份运行。用户之间安全隔离，一个用户出问题不影响其他用户。支持 PHP、ASP、ASP.NET、Java、Ruby 等多种动态开发语言。
- WebSphere：WebSphere Application Server 是功能完善、开放的 Web 应用程序服务器，是 IBM 电子商务计划的核心部分，它是基于 Java 的应用环境，用于建立、部署和管理 Internet 和 Intranet Web 应用程序，适应各种 Web 应用程序服务器的需要。
- WebLogic：WebLogic Server 是一款多功能、基于标准的 Web 应用服务器，为企业构建企业应用提供了坚实的基础。针对各种应用开发、关键性任务的部署，各种系统和数据库的集成、跨 Internet 协作等 Weblogic 都提供了相应的支持。由于它具有全面的功能、对开放标准的遵从性、多层架构、支持基于组件的开发等优势，很多公司的企业级应用都选择它来作为开发和部署的环境。WebLogic Server 在使应用服务器成为企业应用架构的基础方面一直处于领先地位，为构建集成化的企业级应用提供了稳固的基础。
- Apache：目前 Apache 仍然是世界上用得最多的 Web 服务器，其市场占有率很长时间都保持在 60%以上（目前的市场份额约 40%左右）。世界上很多著名的网站都是 Apache 的产物，它的成功之处主要在于它的源代码开放、有一支强大的开发团队、支持跨平台的应用（可以运行在几乎所有的 Unix、Windows、Linux 系统平台上）以及它的可移植性等方面。
- Tomcat：Tomcat 是一个开放源代码、运行 Servlet 和 JSP 的容器。Tomcat 实现了 Servlet 和 JSP 规范。此外，Tomcat 还实现了 Apache-Jakarta 规范而且比绝大多数商业应用软件服务器要好，因此目前也有不少的 Web 服务器都选择了 Tomcat。
- Nginx：读作"engine x"，是一个高性能的 HTTP 和反向代理服务器，也是一个 IMAP/POP3/SMTP 代理服务器。 Nginx 是由 Igor Sysoev 为俄罗斯访问量第二的 Rambler 站点开发的，第一个公开版本 0.1.0 发布于 2004 年 10 月 4 日。其将源代码以类 BSD 许可证的形式发布，因它的稳定性、丰富的功能集、示例配置文件和低系统资源的消耗而闻名。在 2014 年下半年，Nginx 的市场份额达到了 14%。

102、JSP 和 Servlet 是什么关系？
答：其实这个问题在上面已经阐述过了，Servlet 是一个特殊的 Java 程序，它运行于服务器的 JVM 中，能够依靠服务器的支持向浏览器提供显示内容。JSP 本质上是 Servlet 的一种简易形式，JSP 会被服务器处理成一个类似于 Servlet 的 Java 程序，可以简化页面内容的生成。Servlet 和 JSP 最主要的不同点在于，Servlet 的应用逻辑是在 Java 文件中，并且完全从表示层中的 HTML 分离开来。而 JSP 的情况是 Java 和 HTML 可以组合成一个扩展名为.jsp 的文件。有人说，Servlet 就是在 Java 中写 HTML，而 JSP 就是在 HTML 中写 Java 代码，当然这个说法是很片面且不够准确的。JSP 侧重于视图，Servlet 更侧重于控制逻辑，在 MVC 架构模式中，JSP 适合充当视图（view）而 Servlet 适合充当控制器（controller）。

103、讲解 JSP 中的四种作用域。
答：JSP 中的四种作用域包括 page、request、session 和 application，具体来说：
- page 代表与一个页面相关的对象和属性。
- request 代表与 Web 客户机发出的一个请求相关的对象和属性。一个请求可能跨越多个页面，涉及多个 Web 组件；需要在页面显示的临时数据可以置于此作用域。
- session 代表与某个用户与服务器建立的一次会话相关的对象和属性。跟某个用户相关的数据应该放在用户自己的 session 中。
- application 代表与整个 Web 应用程序相关的对象和属性，它实质上是跨越整个 Web 应用

程序，包括多个页面、请求和会话的一个全局作用域。

104、如何实现 JSP 或 Servlet 的单线程模式？
答：
对于 JSP 页面，可以通过 page 指令进行设置。

<%@page isThreadSafe=”false”%>

对于 Servlet，可以让自定义的 Servlet 实现 SingleThreadModel 标识接口。

说明：如果将 JSP 或 Servlet 设置成单线程工作模式，会导致每个请求创建一个 Servlet 实例，这种实践将导致严重的性能问题（服务器的内存压力很大，还会导致频繁的垃圾回收），所以通常情况下并不会这么做。

105、实现会话跟踪的技术有哪些？
答：由于 HTTP 协议本身是无状态的，服务器为了区分不同的用户，就需要对用户会话进行跟踪，简单的说就是为用户进行登记，为用户分配唯一的 ID，下一次用户在请求中包含此 ID，服务器据此判断到底是哪一个用户。
①URL 重写：在 URL 中添加用户会话的信息作为请求的参数，或者将唯一的会话 ID 添加到 URL 结尾以标识一个会话。
②设置表单隐藏域：将和会话跟踪相关的字段添加到隐式表单域中，这些信息不会在浏览器中显示但是提交表单时会提交给服务器。
这两种方式很难处理跨越多个页面的信息传递，因为如果每次都要修改 URL 或在页面中添加隐式表单域来存储用户会话相关信息，事情将变得非常麻烦。
③cookie：cookie 有两种，一种是基于窗口的，浏览器窗口关闭后，cookie 就没有了；另一种是将信息存储在一个临时文件中，并设置存在的时间。当用户通过浏览器和服务器建立一次会话后，会话 ID 就会随响应信息返回存储在基于窗口的 cookie 中，那就意味着只要浏览器没有关闭，会话没有超时，下一次请求时这个会话 ID 又会提交给服务器让服务器识别用户身份。会话中可以为用户保存信息。会话对象是在服务器内存中的，而基于窗口的 cookie 是在客户端内存中的。如果浏览器禁用了 cookie，那么就需要通过下面两种方式进行会话跟踪。当然，在使用 cookie 时要注意几点：首先不要在 cookie 中存放敏感信息；其次 cookie 存储的数据量有限（4k），不能将过多的内容存储 cookie 中；再者浏览器通常只允许一个站点最多存放 20 个 cookie。当然，和用户会话相关的其他信息（除了会话 ID）也可以存在 cookie 方便进行会话跟踪。
④HttpSession：在所有会话跟踪技术中，HttpSession 对象是最强大也是功能最多的。当一个用户第一次访问某个网站时会自动创建 HttpSession，每个用户可以访问他自己的 HttpSession。可以通过 HttpServletRequest 对象的 getSession 方法获得 HttpSession，通过 HttpSession 的 setAttribute 方法可以将一个值放在 HttpSession 中，通过调用 HttpSession 对象的 getAttribute 方法，同时传入属性名就可以获取保存在 HttpSession 中的对象。与上面三种方式不同的是，HttpSession 放在服务器的内存中，因此不要将过大的对象放在里面，即使目前的 Servlet 容器可以在内存将满时将 HttpSession 中的对象移到其他存储设备中，但是这样势必影响性能。添加到 HttpSession 中的值可以是任意 Java 对象，这个对象最好实现了 Serializable 接口，这样 Servlet 容器在必要的时候可以将其序列化到文件中，否则在序列化时就会出现异常。

补充：HTML5 中可以使用 Web Storage 技术通过 JavaScript 来保存数据，例如可以使用 localStorage 和 sessionStorage 来保存用户会话的信息，也能够实现会话跟踪。

106、过滤器有哪些作用和用法？
答： Java Web 开发中的过滤器（filter）是从 Servlet 2.3 规范开始增加的功能，并在 Servlet 2.4 规范中得到增强。对 Web 应用来说，过滤器是一个驻留在服务器端的 Web 组件，它可以截取客户端和服务器之间的请求与响应信息，并对这些信息进行过滤。当 Web 容器接受

到一个对资源的请求时，它将判断是否有过滤器与这个资源相关联。如果有，那么容器将把请求交给过滤器进行处理。在过滤器中，你可以改变请求的内容，或者重新设置请求的报头信息，然后再将请求发送给目标资源。当目标资源对请求作出响应时候，容器同样会将响应先转发给过滤器，在过滤器中你可以对响应的内容进行转换，然后再将响应发送到客户端。

常见的过滤器用途主要包括：对用户请求进行统一认证、对用户的访问请求进行记录和审核、对用户发送的数据进行过滤或替换、转换图象格式、对响应内容进行压缩以减少传输量、对请求或响应进行加解密处理、触发资源访问事件、对 XML 的输出应用 XSLT 等。

和过滤器相关的接口主要有：Filter、FilterConfig 和 FilterChain。

编码过滤器的例子：

```
import java.io.IOException;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.annotation.WebInitParam;

@WebFilter(urlPatterns = { "*" },
      initParams = {@WebInitParam(name="encoding", value="utf-8")})
public class CodingFilter implements Filter {
   private String defaultEncoding = "utf-8";

   @Override
   public void destroy() {
   }

   @Override
   public void doFilter(ServletRequest req, ServletResponse resp,
         FilterChain chain) throws IOException, ServletException {
      req.setCharacterEncoding(defaultEncoding);
      resp.setCharacterEncoding(defaultEncoding);
      chain.doFilter(req, resp);
   }

   @Override
   public void init(FilterConfig config) throws ServletException {
      String encoding = config.getInitParameter("encoding");
      if (encoding != null) {
         defaultEncoding = encoding;
      }
   }
}
```

下载计数过滤器的例子：

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
```

```
import java.io.IOException;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;

@WebFilter(urlPatterns = {"/*"})
public class DownloadCounterFilter implements Filter {

   private ExecutorService executorService = Executors.newSingleThreadExecutor();
   private Properties downloadLog;
   private File logFile;

   @Override
   public void destroy() {
      executorService.shutdown();
   }

   @Override
   public void doFilter(ServletRequest req, ServletResponse resp,
         FilterChain chain) throws IOException, ServletException {
      HttpServletRequest request = (HttpServletRequest) req;
      final String uri = request.getRequestURI();
      executorService.execute(new Runnable() {

         @Override
         public void run() {
            String value = downloadLog.getProperty(uri);
            if(value == null) {
               downloadLog.setProperty(uri, "1");
            }
            else {
               int count = Integer.parseInt(value);
               downloadLog.setProperty(uri, String.valueOf(++count));
            }
            try {
               downloadLog.store(new FileWriter(logFile), "");
            }
            catch (IOException e) {
               e.printStackTrace();
            }
         }
      });
      chain.doFilter(req, resp);
   }

   @Override
   public void init(FilterConfig config) throws ServletException {
      String appPath = config.getServletContext().getRealPath("/");
```

```
        logFile = new File(appPath, "downloadLog.txt");
        if(!logFile.exists()) {
            try {
                logFile.createNewFile();
            }
            catch(IOException e) {
                e.printStackTrace();
            }
        }
        downloadLog = new Properties();
        try {
            downloadLog.load(new FileReader(logFile));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

}
```

说明：这里使用了 Servlet 3 规范中的注解来部署过滤器，当然也可以在 web.xml 中使用<filter>和<filter-mapping>标签部署过滤器，如 108 题中所示。

107、监听器有哪些作用和用法？
答：Java Web 开发中的监听器（listener）就是 application、session、request 三个对象创建、销毁或者往其中添加修改删除属性时自动执行代码的功能组件，如下所示：
①ServletContextListener：对 Servlet 上下文的创建和销毁进行监听。
②ServletContextAttributeListener：监听 Servlet 上下文属性的添加、删除和替换。
③HttpSessionListener：对 Session 的创建和销毁进行监听。

补充：session 的销毁有两种情况：1). session 超时（可以在 web.xml 中通过<session-config>/<session-timeout>标签配置超时时间）；2). 通过调用 session 对象的 invalidate()方法使 session 失效。

④HttpSessionAttributeListener：对 Session 对象中属性的添加、删除和替换进行监听。
⑤ServletRequestListener：对请求对象的初始化和销毁进行监听。
⑥ServletRequestAttributeListener：对请求对象属性的添加、删除和替换进行监听。

下面是一个统计网站最多在线人数监听器的例子。

```
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

/**
 上下文监听器，在服务器启动时初始化 onLineCount 和 maxOnLineCount 两个变量
 并将其置于服务器上下文（ServletContext）中，其初始值都是 0
*/
@WebListener
public class InitListener implements ServletContextListener {

    @Override
    public void contextDestroyed(ServletContextEvent evt) {
    }

    @Override
```

```
    public void contextInitialized(ServletContextEvent evt) {
        evt.getServletContext().setAttribute("onLineCount", 0);
        evt.getServletContext().setAttribute("maxOnLineCount", 0);
    }

}



import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

import javax.servlet.ServletContext;
import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

/**
 会话监听器，在用户会话创建和销毁的时候根据情况
 修改 onLineCount 和 maxOnLineCount 的值
*/
@WebListener
public class MaxCountListener implements HttpSessionListener {

    @Override
    public void sessionCreated(HttpSessionEvent event) {
        ServletContext ctx = event.getSession().getServletContext();
        int count = Integer.parseInt(ctx.getAttribute("onLineCount").toString());
        count++;
        ctx.setAttribute("onLineCount", count);
        int maxOnLineCount = Integer.parseInt(ctx.getAttribute("maxOnLineCount").toString());
        if (count > maxOnLineCount) {
            ctx.setAttribute("maxOnLineCount", count);
            DateFormat df = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            ctx.setAttribute("date", df.format(new Date()));
        }
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent event) {
        ServletContext app = event.getSession().getServletContext();
        int count = Integer.parseInt(app.getAttribute("onLineCount").toString());
        count--;
        app.setAttribute("onLineCount", count);
    }
}
```

说明：这里使用了 Servlet 3 规范中的@WebListener 注解配置监听器，当然你可以在 web.xml 文件中用<listener>标签配置监听器，如 108 题中所示。

108、web.xml 文件中可以配置哪些内容？
答：web.xml 用于配置 Web 应用的相关信息，如：监听器（listener）、过滤器（filter）、Servlet、相关参数、会话超时时间、安全验证方式、错误页面等，下面是一些开发中常见的配置：

①配置 Spring 上下文加载监听器加载 Spring 配置文件并创建 IoC 容器：

```
<context-param>
   <param-name>contextConfigLocation</param-name>
  <param-value>classpath:applicationContext.xml</param-value>
</context-param>

<listener>
   <listener-class>
    org.springframework.web.context.ContextLoaderListener
   </listener-class>
</listener>
```

②配置 Spring 的 OpenSessionInView 过滤器来解决延迟加载和 Hibernate 会话关闭的矛盾：

```
<filter>
    <filter-name>openSessionInView</filter-name>
    <filter-class>
      org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
    </filter-class>
</filter>

<filter-mapping>
    <filter-name>openSessionInView</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

③配置会话超时时间为 10 分钟：

```
<session-config>
    <session-timeout>10</session-timeout>
</session-config>
```

④配置 404 和 Exception 的错误页面：

```
<error-page>
    <error-code>404</error-code>
    <location>/error.jsp</location>
</error-page>

<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/error.jsp</location>
</error-page>
```

⑤配置安全认证方式：

```
<security-constraint>
    <web-resource-collection>
       <web-resource-name>ProtectedArea</web-resource-name>
       <url-pattern>/admin/*</url-pattern>
       <http-method>GET</http-method>
       <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
       <role-name>admin</role-name>
    </auth-constraint>
```

```
  </security-constraint>

  <login-config>
    <auth-method>BASIC</auth-method>
  </login-config>

  <security-role>
    <role-name>admin</role-name>
  </security-role>
```

说明：对 Servlet（小服务）、Listener（监听器）和 Filter（过滤器）等 Web 组件的配置，Servlet 3 规范提供了基于注解的配置方式，可以分别使用@WebServlet、@WebListener、@WebFilter 注解进行配置。

补充：如果 Web 提供了有价值的商业信息或者是敏感数据，那么站点的安全性就是必须考虑的问题。安全认证是实现安全性的重要手段，认证就是要解决“Are you who you say you are?”的问题。认证的方式非常多，简单说来可以分为三类：

A. What you know? — 口令
B. What you have? — 数字证书（U 盾、密保卡）
C. Who you are? — 指纹识别、虹膜识别

在 Tomcat 中可以通过建立安全套接字层（Secure Socket Layer, SSL）以及通过基本验证或表单验证来实现对安全性的支持。

109、你的项目中使用过哪些 JSTL 标签？
答：项目中主要使用了 JSTL 的核心标签库，包括<c:if>、<c:choose>、<c: when>、<c: otherwise>、<c:forEach>等，主要用于构造循环和分支结构以控制显示逻辑。

说明：虽然 JSTL 标签库提供了 core、sql、fmt、xml 等标签库，但是实际开发中建议只使用核心标签库（core），而且最好只使用分支和循环标签并辅以表达式语言（EL），这样才能真正做到数据显示和业务逻辑的分离，这才是最佳实践。

110、使用标签库有什么好处？如何自定义 JSP 标签？
答：使用标签库的好处包括以下几个方面：
- 分离 JSP 页面的内容和逻辑，简化了 Web 开发；
- 开发者可以创建自定义标签来封装业务逻辑和显示逻辑；
- 标签具有很好的可移植性、可维护性和可重用性；
- 避免了对 Scriptlet（小脚本）的使用（很多公司的项目开发都不允许在 JSP 中书写小脚本）

自定义 JSP 标签包括以下几个步骤：
- 编写一个 Java 类实现实现 Tag/BodyTag/IterationTag 接口（开发中通常不直接实现这些接口而是继承 TagSupport/BodyTagSupport/SimpleTagSupport 类，这是对缺省适配模式的应用），重写 doStartTag()、doEndTag()等方法，定义标签要完成的功能
- 编写扩展名为 tld 的标签描述文件对自定义标签进行部署，tld 文件通常放在 WEB-INF 文件夹下或其子目录中
- 在 JSP 页面中使用 taglib 指令引用该标签库

下面是一个自定义标签库的例子。

步骤 1 - 标签类源代码 TimeTag.java：

```
package com.jackfrued.tags;

import java.io.IOException;
```

```
import java.text.SimpleDateFormat;
import java.util.Date;

import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.TagSupport;

public class TimeTag extends TagSupport {
   private static final long serialVersionUID = 1L;

   private String format = "yyyy-MM-dd hh:mm:ss";
   private String foreColor = "black";
   private String backColor = "white";

   public int doStartTag() throws JspException {
       SimpleDateFormat sdf = new SimpleDateFormat(format);
       JspWriter writer = pageContext.getOut();
       StringBuilder sb = new StringBuilder();
       sb.append(String.format("<span style='color:%s;background-color:%s'>%s</span>",
           foreColor, backColor, sdf.format(new Date())));
       try {
         writer.print(sb.toString());
       } catch(IOException e) {
         e.printStackTrace();
       }
       return SKIP_BODY;
     }

   public void setFormat(String format) {
      this.format = format;
   }

   public void setForeColor(String foreColor) {
      this.foreColor = foreColor;
   }

   public void setBackColor(String backColor) {
      this.backColor = backColor;
   }
}
```

步骤 2 - 编写标签库描述文件 my.tld：

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
   http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
   version="2.0">

   <description>定义标签库</description>
   <tlib-version>1.0</tlib-version>
   <short-name>MyTag</short-name>
   <tag>
      <name>time</name>
      <tag-class>com.jackfrued.tags.TimeTag</tag-class>
      <body-content>empty</body-content>
```

```
    <attribute>
      <name>format</name>
      <required>false</required>
    </attribute>
    <attribute>
      <name>foreColor</name>
    </attribute>
    <attribute>
      <name>backColor</name>
    </attribute>
  </tag>
</taglib>
```

步骤 3 - 在 JSP 页面中使用自定义标签：

```
<%@ page pageEncoding="UTF-8"%>
<%@ taglib prefix="my" uri="/WEB-INF/tld/my.tld" %>
<%
String path = request.getContextPath();
String basePath = request.getScheme() + "://" + request.getServerName() + ":" +
request.getServerPort() + path + "/";
%>

<!DOCTYPE html>
<html>
 <head>
  <base href="<%=basePath%>">
  <title>首页</title>
  <style type="text/css">
    * { font-family: "Arial"; font-size:72px; }
  </style>
 </head>

 <body>
  <my:time format="yyyy-MM-dd" backColor="blue" foreColor="yellow"/>
 </body>
</html>
```

提示：如果要将自定义的标签库发布成 JAR 文件，需要将标签库描述文件（tld 文件）放在 JAR 文件的 META-INF 目录下，可以 JDK 中的 jar 工具完成 JAR 文件的生成。

111、说一下表达式语言（EL）的隐式对象及其作用。
答：EL 的隐式对象包括：pageContext、initParam（访问上下文参数）、param（访问请求参数）、paramValues、header（访问请求头）、headerValues、cookie（访问 cookie）、applicationScope（访问 application 作用域）、sessionScope（访问 session 作用域）、requestScope（访问 request 作用域）、pageScope（访问 page 作用域）。

用法如下所示：

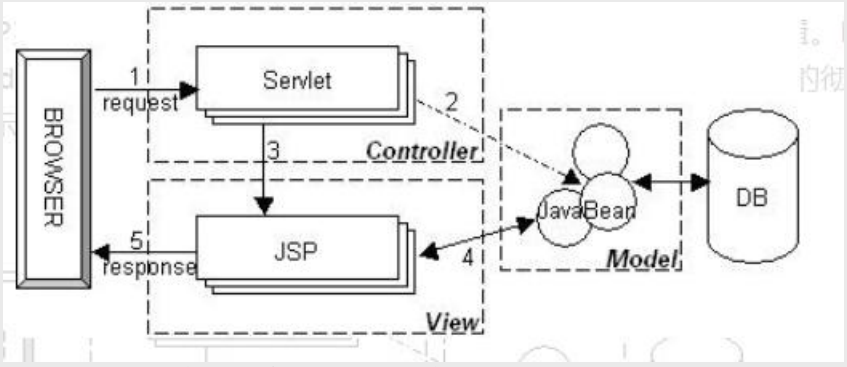
```
${pageContext.request.method}
${pageContext["request"]["method"]}
${pageContext.request["method"]}
${pageContext["request"].method}
${initParam.defaultEncoding}
${header["accept-language"]}
${headerValues["accept-language"][0]}
```

${cookie.jsessionid.value}
${sessionScope.loginUser.username}

补充：表达式语言的.和[]运算作用是一致的，唯一的差别在于如果访问的属性名不符合 Java 标识符命名规则，例如上面的 accept-language 就不是一个有效的 Java 标识符，那么这时候就只能用[]运算符而不能使用.运算符获取它的值

112、表达式语言（EL）支持哪些运算符？
答：除了.和[]运算符，EL 还提供了：
- 算术运算符：+、-、*、/或 div、%或 mod
- 关系运算符：==或 eq、!=或 ne、>或 gt、>=或 ge、<或 lt、<=或 le
- 逻辑运算符：&&或 and、||或 or、!或 not
- 条件运算符：${statement? A : B}（跟 Java 的条件运算符类似）
- empty 运算符：检查一个值是否为 null 或者空（数组长度为 0 或集合中没有元素也返回 true）

113、Java Web 开发的 Model 1 和 Model 2 分别指的是什么？
答：Model 1 是以页面为中心的 Java Web 开发，使用 JSP+JavaBean 技术将页面显示逻辑和业务逻辑处理分开，JSP 实现页面显示，JavaBean 对象用来保存数据和实现业务逻辑。Model 2 是基于 MVC（模型-视图-控制器，Model-View-Controller）架构模式的开发模型，实现了模型和视图的彻底分离，利于团队开发和代码复用，如下图所示。



114、Servlet 3 中的异步处理指的是什么？
答：在 Servlet 3 中引入了一项新的技术可以让 Servlet 异步处理请求。有人可能会质疑，既然都有多线程了，还需要异步处理请求吗？答案是肯定的，因为如果一个任务处理时间相当长，那么 Servlet 或 Filter 会一直占用着请求处理线程直到任务结束，随着并发用户的增加，容器将会遭遇线程超出的风险，这这种情况下很多的请求将会被堆积起来而后续的请求可能会遭遇拒绝服务，直到有资源可以处理请求为止。异步特性可以帮助应用节省容器中的线程，特别适合执行时间长而且用户需要得到结果的任务，如果用户不需要得到结果则直接将一个 Runnable 对象交给 Executor 并立即返回即可。

补充：多线程在 Java 诞生初期无疑是一个亮点，而 Servlet 单实例多线程的工作方式也曾为其赢得美名，然而技术的发展往往会颠覆我们很多的认知，就如同当年爱因斯坦的相对论颠覆了牛顿的经典力学一般。事实上，异步处理绝不是 Serlvet 3 首创，如果你了解 Node.js 的话，对 Servlet 3 的这个重要改进就不以为奇了。

下面是一个支持异步处理请求的 Servlet 的例子。

```
import java.io.IOException;
import javax.servlet.AsyncContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
```

```
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns = {"/async"}, asyncSupported = true)
public class AsyncServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   @Override
   public void doGet(HttpServletRequest req, HttpServletResponse resp)
         throws ServletException, IOException {
      // 开启 Tomcat 异步 Servlet 支持
      req.setAttribute("org.apache.catalina.ASYNC_SUPPORTED", true);

      final AsyncContext ctx = req.startAsync(); // 启动异步处理的上下文
      // ctx.setTimeout(30000);
      ctx.start(new Runnable() {

         @Override
         public void run() {
            // 在此处添加异步处理的代码

            ctx.complete();
         }
      });
   }
}
```

115、如何在基于 Java 的 Web 项目中实现文件上传和下载?
答：在 Sevlet 3 以前，Servlet API 中没有支持上传功能的 API，因此要实现上传功能需要引入第三方工具从 POST 请求中获得上传的附件或者通过自行处理输入流来获得上传的文件，我们推荐使用 Apache 的 commons-fileupload。
从 Servlet 3 开始，文件上传变得无比简单，相信看看下面的例子一切都清楚了。
上传页面 index.jsp：

```
<%@ page pageEncoding="utf-8"%>
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Photo Upload</title>
</head>
<body>
<h1>Select your photo and upload</h1>
<hr/>
<div style="color:red;font-size:14px;">${hint}</div>
<form action="UploadServlet" method="post" enctype="multipart/form-data">
   Photo file: <input type="file" name="photo" />
   <input type="submit" value="Upload" />
</form>
</body>
</html>
```

支持上传的 Servlet：

```
package com.jackfrued.servlet;
```

```
import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.annotation.MultipartConfig;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.Part;

@WebServlet("/UploadServlet")
@MultipartConfig
public class UploadServlet extends HttpServlet {
   private static final long serialVersionUID = 1L;

   protected void doPost(HttpServletRequest request,
         HttpServletResponse response) throws ServletException, IOException {
      // 可以用 request.getPart()方法获得名为 photo 的上传附件
      // 也可以用 request.getParts()获得所有上传附件（多文件上传）
      // 然后通过循环分别处理每一个上传的文件
      Part part = request.getPart("photo");
      if (part != null && part.getSubmittedFileName().length() > 0) {
         // 用 ServletContext 对象的 getRealPath()方法获得上传文件夹的绝对路径
         String savePath = request.getServletContext().getRealPath("/upload");
         // Servlet 3.1 规范中可以用 Part 对象的 getSubmittedFileName()方法获得上传的文件
名
         // 更好的做法是为上传的文件进行重命名（避免同名文件的相互覆盖）
         part.write(savePath + "/" + part.getSubmittedFileName());
         request.setAttribute("hint", "Upload Successfully!");
      } else {
         request.setAttribute("hint", "Upload failed!");
      }
      // 跳转回到上传页面
      request.getRequestDispatcher("index.jsp").forward(request, response);
   }

}
```

116、服务器收到用户提交的表单数据，到底是调用 Servlet 的 doGet()还是 doPost()方法？
答：HTML 的<form>元素有一个 method 属性，用来指定提交表单的方式，其值可以是 get 或 post。我们自定义的 Servlet 一般情况下会重写 doGet()或 doPost()两个方法之一或全部，如果是 GET 请求就调用 doGet()方法，如果是 POST 请求就调用 doPost()方法，那为什么为什么这样呢？我们自定义的 Servlet 通常继承自 HttpServlet，HttpServlet 继承自 GenericServlet 并重写了其中的 service()方法，这个方法是 Servlet 接口中定义的。HttpServlet 重写的 service()方法会先获取用户请求的方法，然后根据请求方法调用 doGet()、doPost()、doPut()、doDelete()等方法，如果在自定义 Servlet 中重写了这些方法，那么显然会调用重写过的（自定义的）方法，这显然是对模板方法模式的应用（如果不理解，请参考阎宏博士的《Java 与模式》一书的第 37 章）。当然，自定义 Servlet 中也可以直接重写 service()方法，那么不管是哪种方式的请求，都可以通过自己的代码进行处理，这对于不区分请求方法的场景比较合适。

117、JSP 中的静态包含和动态包含有什么区别？
答：静态包含是通过 JSP 的 include 指令包含页面，动态包含是通过 JSP 标准动作

<jsp:forward>包含页面。静态包含是编译时包含，如果包含的页面不存在则会产生编译错误，而且两个页面的"contentType"属性应保持一致，因为两个页面会合二为一，只产生一个 class 文件，因此被包含页面发生的变动再包含它的页面更新前不会得到更新。动态包含是运行时包含，可以向被包含的页面传递参数，包含页面和被包含页面是独立的，会编译出两个 class 文件，如果被包含的页面不存在，不会产生编译错误，也不影响页面其他部分的执行。代码如下所示：

```
<%-- 静态包含 --%>
<%@ include file="..." %>

<%-- 动态包含 --%>
<jsp:include page="...">
   <jsp:param name="..." value="..." />
</jsp:include>
```

118、Servlet 中如何获取用户提交的查询参数或表单数据？
答：可以通过请求对象（HttpServletRequest）的 getParameter()方法通过参数名获得参数值。如果有包含多个值的参数（例如复选框），可以通过请求对象的 getParameterValues()方法获得。当然也可以通过请求对象的 getParameterMap()获得一个参数名和参数值的映射（Map）。

119、Servlet 中如何获取用户配置的初始化参数以及服务器上下文参数？
答：可以通过重写 Servlet 接口的 init(ServletConfig)方法并通过 ServletConfig 对象的 getInitParameter()方法来获取 Servlet 的初始化参数。可以通过 ServletConfig 对象的 getServletContext()方法获取 ServletContext 对象，并通过该对象的 getInitParameter()方法来获取服务器上下文参数。当然，ServletContext 对象也在处理用户请求的方法（如 doGet()方法）中通过请求对象的 getServletContext()方法来获得。

120、如何设置请求的编码以及响

应内容的类型？
答：通过请求对象（ServletRequest）的 setCharacterEncoding(String)方法可以设置请求的编码，其实要彻底解决乱码问题就应该让页面、服务器、请求和响应、Java 程序都使用统一的编码，最好的选择当然是 UTF-8；通过响应对象（ServletResponse）的 setContentType(String)方法可以设置响应内容的类型，当然也可以通过 HttpServletResponsed 对象的 setHeader(String, String)方法来设置。
说明：现在如果还有公司在面试的时候问 JSP 的声明标记、表达式标记、小脚本标记这些内容的话，这样的公司也不用去了，其实 JSP 内置对象、JSP 指令这些东西基本上都可以忘却了，关于 Java Web 开发的相关知识，可以看一下我的《Servlet&JSP 思维导图》，上面有完整的知识点的罗列。想了解如何实现自定义 MVC 框架的，可以看一下我的《Java Web 自定义 MVC 框架详解》。

121、解释一下网络应用的模式及其特点。
答：典型的网络应用模式大致有三类：B/S、C/S、P2P。其中 B 代表浏览器（Browser）、C 代表客户端（Client）、S 代表服务器（Server），P2P 是对等模式，不区分客户端和服务器。B/S 应用模式中可以视为特殊的 C/S 应用模式，只是将 C/S 应用模式中的特殊的客户端换成了浏览器，因为几乎所有的系统上都有浏览器，那么只要打开浏览器就可以使用应用，没有安装、配置、升级客户端所带来的各种开销。P2P 应用模式中，成千上万台彼此连接的计算机都处于对等的地位，整个网络一般来说不依赖专用的集中服务器。网络中的每一台计算机既能充当网络服务的请求者，又对其它计算机的请求作出响应，提供资源和服务。通常这些资源和服务包括：信息的共享和交换、计算资源（如 CPU 的共享）、存储共享（如缓存和磁盘空间的使用）等，这种应用模式最大的阻力安全性、版本等问题，目

前有很多应用都混合使用了多种应用模型，最常见的网络视频应用，它几乎把三种模式都用上了。

补充：此题要跟"电子商务模式"区分开，因为有很多人被问到这个问题的时候马上想到的是 B2B（如阿里巴巴）、B2C（如当当、亚马逊、京东）、C2C（如淘宝、拍拍）、C2B（如威客）、O2O（如美团、饿了么）。对于这类问题，可以去百度上面科普一下。

122、什么是 Web Service（Web 服务）？
答：从表面上看，Web Service 就是一个应用程序，它向外界暴露出一个能够通过 Web 进行调用的 API。这就是说，你能够用编程的方法透明的调用这个应用程序，不需要了解它的任何细节，跟你使用的编程语言也没有关系。例如可以创建一个提供天气预报的 Web Service，那么无论你用哪种编程语言开发的应用都可以通过调用它的 API 并传入城市信息来获得该城市的天气预报。之所以称之为 Web Service，是因为它基于 HTTP 协议传输数据，这使得运行在不同机器上的不同应用无须借助附加的、专门的第三方软件或硬件，就可相互交换数据或集成。

补充：这里必须要提及的一个概念是 SOA（Service-Oriented Architecture，面向服务的架构），SOA 是一种思想，它将应用程序的不同功能单元通过中立的契约联系起来，独立于硬件平台、操作系统和编程语言，使得各种形式的功能单元能够更好的集成。显然，Web Service 是 SOA 的一种较好的解决方案，它更多的是一种标准，而不是一种具体的技术。

123、概念解释：SOAP、WSDL、UDDI。
答：
- SOAP：简单对象访问协议（Simple Object Access Protocol），是 Web Service 中交换数据的一种协议规范。
- WSDL：Web 服务描述语言（Web Service Description Language），它描述了 Web 服务的公共接口。这是一个基于 XML 的关于如何与 Web 服务通讯和使用的服务描述；也就是描述与目录中列出的 Web 服务进行交互时需要绑定的协议和信息格式。通常采用抽象语言描述该服务支持的操作和信息，使用的时候再将实际的网络协议和信息格式绑定给该服务。
- UDDI：统一描述、发现和集成（Universal Description, Discovery and Integration），它是一个基于 XML 的跨平台的描述规范，可以使世界范围内的企业在互联网上发布自己所提供的服务。简单的说，UDDI 是访问各种 WSDL 的一个门面（可以参考设计模式中的门面模式）。

提示：关于 Web Service 的相关概念和知识可以在 W3CSchool 上找到相关的资料。

124、Java 规范中和 Web Service 相关的规范有哪些？
答：Java 规范中和 Web Service 相关的有三个：
- JAX-WS(JSR 224)：这个规范是早期的基于 SOAP 的 Web Service 规范 JAX-RPC 的替代版本，它并不提供向下兼容性，因为 RPC 样式的 WSDL 以及相关的 API 已经在 Java EE5 中被移除了。WS-MetaData 是 JAX-WS 的依赖规范，提供了基于注解配置 Web Service 和 SOAP 消息的相关 API。
- JAXM(JSR 67)：定义了发送和接收消息所需的 API,相当于 Web Service 的服务器端。
- JAX-RS(JSR 311 & JSR 339 & JSR 370)：是 Java 针对 REST（Representation State Transfer）架构风格制定的一套 Web Service 规范。REST 是一种软件架构模式，是一种风格，它不像 SOAP 那样本身承载着一种消息协议，(两种风格的 Web Service 均采用了 HTTP 做传输协议，因为 HTTP 协议能穿越防火墙，Java 的远程方法调用（RMI）等是重量级协议，通常不能穿越防火墙），因此可以将 REST 视为基于 HTTP 协议的软件架构。REST 中最重要的两个概念是资源定位和资源操作，而 HTTP 协议恰好完整的提供了这两个点。HTTP 协议中的 URI 可以完成资源定位，而 GET、POST、OPTION、DELETE 方法可以完成资源操作。因此 REST 完全依赖 HTTP 协议就可以完成 Web Service，而不像 SOAP 协议那样只利用了

HTTP 的传输特性，定位和操作都是由 SOAP 协议自身完成的，也正是由于 SOAP 消息的存在使得基于 SOAP 的 Web Service 显得笨重而逐渐被淘汰。

125、介绍一下你了解的 Java 领域的 Web Service 框架。
答：Java 领域的 Web Service 框架很多，包括 Axis2（Axis 的升级版本）、Jersey（RESTful 的 Web Service 框架）、CXF（XFire 的延续版本）、Hessian、Turmeric、JBoss SOA 等，其中绝大多数都是开源框架。

提示：面试被问到这类问题的时候一定选择自己用过的最熟悉的作答，如果之前没有了解过就应该在面试前花一些时间了解其中的两个，并比较其优缺点，这样才能在面试时给出一个漂亮的答案。

这部分主要是开源 Java EE 框架方面的内容，包括 Hibernate、MyBatis、Spring、Spring MVC 等，由于 Struts 2 已经是明日黄花，在这里就不讨论 Struts 2 的面试题，如果需要了解相关内容，可以参考 Java 程序员面试题集（86-115）。

126、什么是 ORM？
答：对象关系映射（Object-Relational Mapping，简称 ORM）是一种为了解决程序的面向对象模型与数据库的关系模型互不匹配问题的技术；简单的说，ORM 是通过使用描述对象和数据库之间映射的元数据（在 Java 中可以用 XML 或者是注解），将程序中的对象自动持久化到关系数据库中或者将关系数据库表中的行转换成 Java 对象，其本质上就是将数据从一种形式转换到另外一种形式。

127、持久层设计要考虑的问题有哪些？你用过的持久层框架有哪些？
答：所谓"持久"就是将数据保存到可掉电式存储设备中以便今后使用，简单的说，就是将内存中的数据保存到关系型数据库、文件系统、消息队列等提供持久化支持的设备中。持久层就是系统中专注于实现数据持久化的相对独立的层面。

持久层设计的目标包括：
- 数据存储逻辑的分离，提供抽象化的数据访问接口。
- 数据访问底层实现的分离，可以在不修改代码的情况下切换底层实现。
- 资源管理和调度的分离，在数据访问层实现统一的资源调度（如缓存机制）。
- 数据抽象，提供更面向对象的数据操作。

持久层框架有：
- Hibernate
- MyBatis
- TopLink
- Guzz
- jOOQ
- Spring Data
- ActiveJDBC

128、Hibernate 中 SessionFactory 是线程安全的吗？Session 是线程安全的吗（两个线程能够共享同一个 Session 吗）？
答：SessionFactory 对应 Hibernate 的一个数据存储的概念，它是线程安全的，可以被多个线程并发访问。SessionFactory 一般只会在启动的时候构建。对于应用程序，最好将 SessionFactory 通过单例模式进行封装以便于访问。Session 是一个轻量级非线程安全的对象（线程间不能共享 session），它表示与数据库进行交互的一个工作单元。Session 是由 SessionFactory 创建的，在任务完成之后它会被关闭。Session 是持久层服务对外提供的主要接口。Session 会延迟获取数据库连接（也就是在需要的时候才会获取）。为了避免创建太多的 session，可以使用 ThreadLocal 将 session 和当前线程绑定在一起，这样可以让同一

个线程获得的总是同一个 session。Hibernate 3 中 SessionFactory 的 getCurrentSession()方法就可以做到。

129、Hibernate 中 Session 的 load 和 get 方法的区别是什么？
答：主要有以下三项区别：
① 如果没有找到符合条件的记录，get 方法返回 null，load 方法抛出异常。
② get 方法直接返回实体类对象，load 方法返回实体类对象的代理。
③ 在 Hibernate 3 之前，get 方法只在一级缓存中进行数据查找，如果没有找到对应的数据则越过二级缓存，直接发出 SQL 语句完成数据读取；load 方法则可以从二级缓存中获取数据；从 Hibernate 3 开始，get 方法不再是对二级缓存只写不读，它也是可以访问二级缓存的。

说明：对于 load()方法 Hibernate 认为该数据在数据库中一定存在可以放心的使用代理来实现延迟加载，如果没有数据就抛出异常，而通过 get()方法获取的数据可以不存在。

130、Session 的 save()、update()、merge()、lock()、saveOrUpdate()和 persist()方法分别是做什么的？有什么区别？
答：Hibernate 的对象有三种状态：瞬时态（transient）、持久态（persistent）和游离态（detached），如第 135 题中的图所示。瞬时态的实例可以通过调用 save()、persist()或者 saveOrUpdate()方法变成持久态；游离态的实例可以通过调用 update()、saveOrUpdate()、lock()或者 replicate()变成持久态。save()和 persist()将会引发 SQL 的 INSERT 语句，而 update()或 merge()会引发 UPDATE 语句。save()和 update()的区别在于一个是将瞬时态对象变成持久态，一个是将游离态对象变为持久态。merge()方法可以完成 save()和 update()方法的功能，它的意图是将新的状态合并到已有的持久化对象上或创建新的持久化对象。对于 persist()方法，按照官方文档的说明：① persist()方法把一个瞬时态的实例持久化，但是并不保证标识符被立刻填入到持久化实例中，标识符的填入可能被推迟到 flush 的时间；② persist()方法保证当它在一个事务外部被调用的时候并不触发一个 INSERT 语句，当需要封装一个长会话流程的时候，persist()方法是很有必要的；③ save()方法不保证第②条，它要返回标识符，所以它会立即执行 INSERT 语句，不管是在事务内部还是外部。至于 lock()方法和 update()方法的区别，update()方法是把一个已经更改过的脱管状态的对象变成持久状态；lock()方法是把一个没有更改过的脱管状态的对象变成持久状态。

131、阐述 Session 加载实体对象的过程。
答：Session 加载实体对象的步骤是：
① Session 在调用数据库查询功能之前，首先会在一级缓存中通过实体类型和主键进行查找，如果一级缓存查找命中且数据状态合法，则直接返回；
② 如果一级缓存没有命中，接下来 Session 会在当前 NonExists 记录（相当于一个查询黑名单，如果出现重复的无效查询可以迅速做出判断，从而提升性能）中进行查找，如果 NonExists 中存在同样的查询条件，则返回 null；
③ 如果一级缓存查询失败则查询二级缓存，如果二级缓存命中则直接返回；
④ 如果之前的查询都未命中，则发出 SQL 语句，如果查询未发现对应记录则将此次查询添加到 Session 的 NonExists 中加以记录，并返回 null；
⑤ 根据映射配置和 SQL 语句得到 ResultSet，并创建对应的实体对象；
⑥ 将对象纳入 Session（一级缓存）的管理；
⑦ 如果有对应的拦截器，则执行拦截器的 onLoad 方法；
⑧ 如果开启并设置了要使用二级缓存，则将数据对象纳入二级缓存；
⑨ 返回数据对象。

132、Query 接口的 list 方法和 iterate 方法有什么区别？
答：
① list()方法无法利用一级缓存和二级缓存（对缓存只写不读），它只能在开启查询缓存的

前提下使用查询缓存；iterate()方法可以充分利用缓存，如果目标数据只读或者读取频繁，使用 iterate()方法可以减少性能开销。
② list()方法不会引起 N+1 查询问题，而 iterate()方法可能引起 N+1 查询问题

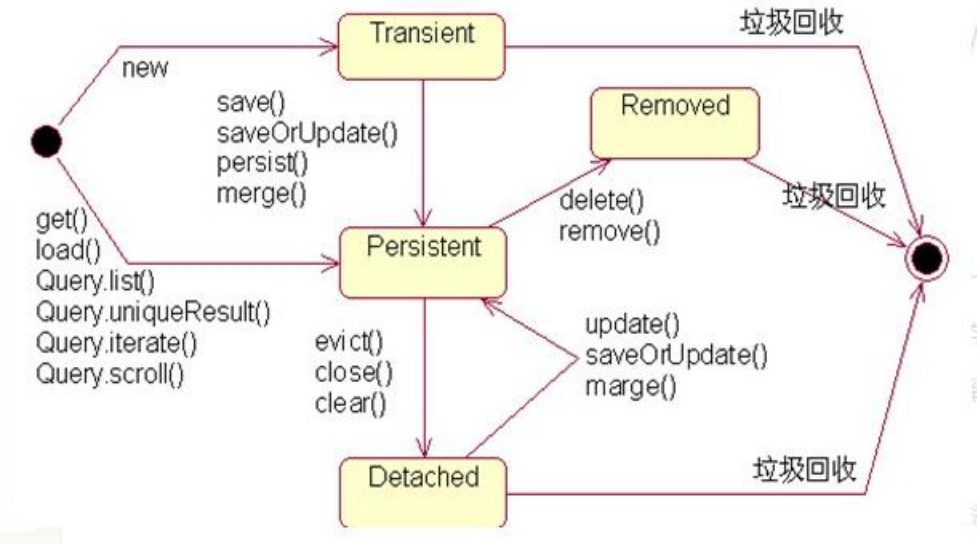
说明：关于 N+1 查询问题，可以参考 CSDN 上的一篇文章《什么是 N+1 查询》

133、Hibernate 如何实现分页查询？
答：通过 Hibernate 实现分页查询，开发人员只需要提供 HQL 语句（调用 Session 的 createQuery()方法）或查询条件（调用 Session 的 createCriteria()方法）、设置查询起始行数（调用 Query 或 Criteria 接口的 setFirstResult()方法）和最大查询行数（调用 Query 或 Criteria 接口的 setMaxResults()方法），并调用 Query 或 Criteria 接口的 list()方法，Hibernate 会自动生成分页查询的 SQL 语句。

134、锁机制有什么用？简述 Hibernate 的悲观锁和乐观锁机制。
答：有些业务逻辑在执行过程中要求对数据进行排他性的访问，于是需要通过一些机制保证在此过程中数据被锁住不会被外界修改，这就是所谓的锁机制。
Hibernate 支持悲观锁和乐观锁两种锁机制。悲观锁，顾名思义悲观的认为在数据处理过程中极有可能存在修改数据的并发事务（包括本系统的其他事务或来自外部系统的事务），于是将处理的数据设置为锁定状态。悲观锁必须依赖数据库本身的锁机制才能真正保证数据访问的排他性，关于数据库的锁机制和事务隔离级别在《Java 面试题大全（上）》中已经讨论过了。乐观锁，顾名思义，对并发事务持乐观态度（认为对数据的并发操作不会经常性的发生），通过更加宽松的锁机制来解决由于悲观锁排他性的数据访问对系统性能造成的严重影响。最常见的乐观锁是通过数据版本标识来实现的，读取数据时获得数据的版本号，更新数据时将此版本号加 1，然后和数据库表对应记录的当前版本号进行比较，如果提交的数据版本号大于数据库中此记录的当前版本号则更新数据，否则认为是过期数据无法更新。Hibernate 中通过 Session 的 get()和 load()方法从数据库中加载对象时可以通过参数指定使用悲观锁；而乐观锁可以通过给实体类加整型的版本字段再通过 XML 或 @Version 注解进行配置。

提示：使用乐观锁会增加了一个版本字段，很明显这需要额外的空间来存储这个版本字段，浪费了空间，但是乐观锁会让系统具有更好的并发性，这是对时间的节省。因此乐观锁也是典型的空间换时间的策略。

135、阐述实体对象的三种状态以及转换关系。
答：最新的 Hibernate 文档中为 Hibernate 对象定义了四种状态（原来是三种状态，面试的时候基本上问的也是三种状态），分别是：瞬时态（new, or transient）、持久态（managed, or persistent）、游状态（detached）和移除态（removed，以前 Hibernate 文档中定义的三种状态中没有移除态），如下图所示，就以前的 Hibernate 文档中移除态被视为是瞬时态。



瞬时态：当 new 一个实体对象后，这个对象处于瞬时态，即这个对象只是一个保存临时数据的内存区域，如果没有变量引用这个对象，则会被 JVM 的垃圾回收机制回收。这个对象所保存的数据与数据库没有任何关系，除非通过 Session 的 save()、saveOrUpdate()、persist()、merge()方法把瞬时态对象与数据库关联，并把数据插入或者更新到数据库，这个对象才转换为持久态对象。

持久态：持久态对象的实例在数据库中有对应的记录，并拥有一个持久化标识（ID）。对持久态对象进行 delete 操作后，数据库中对应的记录将被删除，那么持久态对象与数据库记录不再存在对应关系，持久态对象变成移除态（可以视为瞬时态）。持久态对象被修改变更后，不会马上同步到数据库，直到数据库事务提交。

游离态：当 Session 进行了 close()、clear()、evict()或 flush()后，实体对象从持久态变成游离态，对象虽然拥有持久和与数据库对应记录一致的标识值，但是因为对象已经从会话中清除掉，对象不在持久化管理之内，所以处于游离态（也叫脱管态）。游离态的对象与临时状态对象是十分相似的，只是它还含有持久化标识。

提示：关于这个问题，在 Hibernate 的官方文档中有更为详细的解读。

136、如何理解 Hibernate 的延迟加载机制？在实际应用中，延迟加载与 Session 关闭的矛盾是如何处理的？

答：延迟加载就是并不是在读取的时候就把数据加载进来，而是等到使用时再加载。Hibernate 使用了虚拟代理机制实现延迟加载，我们使用 Session 的 load()方法加载数据或者一对多关联映射在使用延迟加载的情况下从一的一方加载多的一方，得到的都是虚拟代理，简单的说返回给用户的并不是实体本身，而是实体对象的代理。代理对象在用户调用 getter 方法时才会去数据库加载数据。但加载数据就需要数据库连接。而当我们把会话关闭时，数据库连接就同时关闭了。

延迟加载与 session 关闭的矛盾一般可以这样处理：

① 关闭延迟加载特性。这种方式操作起来比较简单，因为 Hibernate 的延迟加载特性是可以通过映射文件或者注解进行配置的，但这种解决方案存在明显的缺陷。首先，出现"no session or session was closed"通常说明系统中已经存在主外键关联，如果去掉延迟加载的话，每次查询的开销都会变得很大。

② 在 session 关闭之前先获取需要查询的数据，可以使用工具方法 Hibernate.isInitialized()判断对象是否被加载，如果没有被加载则可以使用 Hibernate.initialize()方法加载对象。

③ 使用拦截器或过滤器延长 Session 的生命周期直到视图获得数据。Spring 整合 Hibernate 提供的 OpenSessionInViewFilter 和 OpenSessionInViewInterceptor 就是这种做法。

137、举一个多对多关联的例子，并说明如何实现多对多关联映射。
答：例如：商品和订单、学生和课程都是典型的多对多关系。可以在实体类上通过@ManyToMany 注解配置多对多关联或者通过映射文件中的和标签配置多对多关联，但是实际项目开发中，很多时候都是将多对多关联映射转换成两个多对一关联映射来实现的。

138、谈一下你对继承映射的理解。
答：继承关系的映射策略有三种：
① 每个继承结构一张表（table per class hierarchy），不管多少个子类都用一张表。
② 每个子类一张表（table per subclass），公共信息放一张表，特有信息放单独的表。
③ 每个具体类一张表（table per concrete class），有多少个子类就有多少张表。
第一种方式属于单表策略，其优点在于查询子类对象的时候无需表连接，查询速度快，适合多态查询；缺点是可能导致表很大。后两种方式属于多表策略，其优点在于数据存储紧凑，其缺点是需要进行连接查询，不适合多态查询。

139、简述 Hibernate 常见优化策略。
答：这个问题应当挑自己使用过的优化策略回答，常用的有：
① 制定合理的缓存策略（二级缓存、查询缓存）。
② 采用合理的 Session 管理机制。
③ 尽量使用延迟加载特性。
④ 设定合理的批处理参数。
⑤ 如果可以，选用 UUID 作为主键生成器。
⑥ 如果可以，选用基于版本号的乐观锁替代悲观锁。
⑦ 在开发过程中, 开启 hibernate.show_sql 选项查看生成的 SQL，从而了解底层的状况；开发完成后关闭此选项。
⑧ 考虑数据库本身的优化，合理的索引、恰当的数据分区策略等都会对持久层的性能带来可观的提升，但这些需要专业的 DBA（数据库管理员）提供支持。

140、谈一谈 Hibernate 的一级缓存、二级缓存和查询缓存。
答：Hibernate 的 Session 提供了一级缓存的功能，默认总是有效的，当应用程序保存持久化实体、修改持久化实体时，Session 并不会立即把这种改变提交到数据库，而是缓存在当前的 Session 中，除非显示调用了 Session 的 flush()方法或通过 close()方法关闭 Session。通过一级缓存，可以减少程序与数据库的交互，从而提高数据库访问性能。
SessionFactory 级别的二级缓存是全局性的，所有的 Session 可以共享这个二级缓存。不过二级缓存默认是关闭的，需要显示开启并指定需要使用哪种二级缓存实现类（可以使用第三方提供的实现）。一旦开启了二级缓存并设置了需要使用二级缓存的实体类，SessionFactory 就会缓存访问过的该实体类的每个对象，除非缓存的数据超出了指定的缓存空间。
一级缓存和二级缓存都是对整个实体进行缓存，不会缓存普通属性，如果希望对普通属性进行缓存，可以使用查询缓存。查询缓存是将 HQL 或 SQL 语句以及它们的查询结果作为键值对进行缓存，对于同样的查询可以直接从缓存中获取数据。查询缓存默认也是关闭的，需要显示开启。

141、Hibernate 中 DetachedCriteria 类是做什么的？
答：DetachedCriteria 和 Criteria 的用法基本上是一致的，但 Criteria 是由 Session 的 createCriteria()方法创建的，也就意味着离开创建它的 Session，Criteria 就无法使用了。DetachedCriteria 不需要 Session 就可以创建（使用 DetachedCriteria.forClass()方法创建），所以通常也称其为离线的 Criteria，在需要进行查询操作的时候再和 Session 绑定（调用其 getExecutableCriteria(Session)方法），这也就意味着一个 DetachedCriteria 可以在需要的时候和不同的 Session 进行绑定。

142、@OneToMany 注解的 mappedBy 属性有什么作用？

答：@OneToMany 用来配置一对多关联映射，但通常情况下，一对多关联映射都由多的一方来维护关联关系，例如学生和班级，应该在学生类中添加班级属性来维持学生和班级的关联关系（在数据库中是由学生表中的外键班级编号来维护学生表和班级表的多对一关系），如果要使用双向关联，在班级类中添加一个容器属性来存放学生，并使用 @OneToMany 注解进行映射，此时 mappedBy 属性就非常重要。如果使用 XML 进行配置，可以用<set>标签的 inverse="true"设置来达到同样的效果。

143、MyBatis 中使用#和$书写占位符有什么区别？
答：#将传入的数据都当成一个字符串，会对传入的数据自动加上引号；$将传入的数据直接显示生成在 SQL 中。注意：使用$占位符可能会导致 SQL 注射攻击，能用#的地方就不要使用$，写 order by 子句的时候应该用$而不是#。

144、解释一下 MyBatis 中命名空间（namespace）的作用。
答：在大型项目中，可能存在大量的 SQL 语句，这时候为每个 SQL 语句起一个唯一的标识（ID）就变得并不容易了。为了解决这个问题，在 MyBatis 中，可以为每个映射文件起一个唯一的命名空间，这样定义在这个映射文件中的每个 SQL 语句就成了定义在这个命名空间中的一个 ID。只要我们能够保证每个命名空间中这个 ID 是唯一的，即使在不同映射文件中的语句 ID 相同，也不会再产生冲突了。

145、MyBatis 中的动态 SQL 是什么意思？
答：对于一些复杂的查询，我们可能会指定多个查询条件，但是这些条件可能存在也可能不存在，例如在 58 同城上面找房子，我们可能会指定面积、楼层和所在位置来查找房源，也可能会指定面积、价格、户型和所在位置来查找房源，此时就需要根据用户指定的条件动态生成 SQL 语句。如果不使用持久层框架我们可能需要自己拼装 SQL 语句，还好 MyBatis 提供了动态 SQL 的功能来解决这个问题。MyBatis 中用于实现动态 SQL 的元素主要有：
- if
- choose / when / otherwise
- trim
- where
- set
- foreach

下面是映射文件的片段。

```
<select id="foo" parameterType="Blog" resultType="Blog">
      select * from t_blog where 1 = 1
      <if test="title != null">
        and title = #{title}
      </if>
      <if test="content != null">
        and content = #{content}
      </if>
      <if test="owner != null">
        and owner = #{owner}
      </if>
  </select>
```

当然也可以像下面这些书写。

```
<select id="foo" parameterType="Blog" resultType="Blog">
      select * from t_blog where 1 = 1
      <choose>
        <when test="title != null">
```

```
            and title = #{title}
        </when>
        <when test="content != null">
            and content = #{content}
        </when>
        <otherwise>
            and owner = "owner1"
        </otherwise>
    </choose>
</select>
```

再看看下面这个例子。

```
<select id="bar" resultType="Blog">
    select * from t_blog where id in
    <foreach collection="array" index="index"
        item="item" open="(" separator="," close=")">
        #{item}
    </foreach>
</select>
```

146、什么是 IoC 和 DI？DI 是如何实现的？
答：IoC 叫控制反转，是 Inversion of Control 的缩写，DI（Dependency Injection）叫依赖注入，是对 IoC 更简单的诠释。控制反转是把传统上由程序代码直接操控的对象的调用权交给容器，通过容器来实现对象组件的装配和管理。所谓的"控制反转"就是对组件对象控制权的转移，从程序代码本身转移到了外部容器，由容器来创建对象并管理对象之间的依赖关系。IoC 体现了好莱坞原则 - "Don’t call me, we will call you"。依赖注入的基本原则是应用组件不应该负责查找资源或者其他依赖的协作对象。配置对象的工作应该由容器负责，查找资源的逻辑应该从应用组件的代码中抽取出来，交给容器来完成。DI 是对 IoC 更准确的描述，即组件之间的依赖关系由容器在运行期决定，形象的来说，即由容器动态的将某种依赖关系注入到组件之中。

举个例子：一个类 A 需要用到接口 B 中的方法，那么就需要为类 A 和接口 B 建立关联或依赖关系，最原始的方法是在类 A 中创建一个接口 B 的实现类 C 的实例，但这种方法需要开发人员自行维护二者的依赖关系，也就是说当依赖关系发生变动的时候需要修改代码并重新构建整个系统。如果通过一个容器来管理这些对象以及对象的依赖关系，则只需要在类 A 中定义好用于关联接口 B 的方法（构造器或 setter 方法），将类 A 和接口 B 的实现类 C 放入容器中，通过对容器的配置来实现二者的关联。

依赖注入可以通过 setter 方法注入（设值注入）、构造器注入和接口注入三种方式来实现，Spring 支持 setter 注入和构造器注入，通常使用构造器注入来注入必须的依赖关系，对于可选的依赖关系，则 setter 注入是更好的选择，setter 注入需要类提供无参构造器或者无参的静态工厂方法来创建对象。

147、Spring 中 Bean 的作用域有哪些？
答：在 Spring 的早期版本中，仅有两个作用域：singleton 和 prototype，前者表示 Bean 以单例的方式存在；后者表示每次从容器中调用 Bean 时，都会返回一个新的实例，prototype 通常翻译为原型。

补充：设计模式中的创建型模式中也有一个原型模式，原型模式也是一个常用的模式，例如做一个室内设计软件，所有的素材都在工具箱中，而每次从工具箱中取出的都是素材对象的一个原型，可以通过对象克隆来实现原型模式。

Spring 2.x 中针对 WebApplicationContext 新增了 3 个作用域，分别是：request（每次 HTTP

请求都会创建一个新的 Bean）、session（同一个 HttpSession 共享同一个 Bean，不同的 HttpSession 使用不同的 Bean）和 globalSession（同一个全局 Session 共享一个 Bean）。

说明：单例模式和原型模式都是重要的设计模式。一般情况下，无状态或状态不可变的类适合使用单例模式。在传统开发中，由于 DAO 持有 Connection 这个非线程安全对象因而没有使用单例模式；但在 Spring 环境下，所有 DAO 类对可以采用单例模式，因为 Spring 利用 AOP 和 Java API 中的 ThreadLocal 对非线程安全的对象进行了特殊处理。

ThreadLocal 为解决多线程程序的并发问题提供了一种新的思路。ThreadLocal，顾名思义是线程的一个本地化对象，当工作于多线程中的对象使用 ThreadLocal 维护变量时，ThreadLocal 为每个使用该变量的线程分配一个独立的变量副本，所以每一个线程都可以独立的改变自己的副本，而不影响其他线程所对应的副本。从线程的角度看，这个变量就像是线程的本地变量。

ThreadLocal 类非常简单好用，只有四个方法，能用上的也就是下面三个方法：
- void set(T value)：设置当前线程的线程局部变量的值。
- T get()：获得当前线程所对应的线程局部变量的值。
- void remove()：删除当前线程中线程局部变量的值。

ThreadLocal 是如何做到为每一个线程维护一份独立的变量副本的呢？在 ThreadLocal 类中有一个 Map，键为线程对象，值是其线程对应的变量的副本，自己要模拟实现一个 ThreadLocal 类其实并不困难，代码如下所示：

```
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class MyThreadLocal<T> {
    private Map<Thread, T> map = Collections.synchronizedMap(new HashMap<Thread, T>());

    public void set(T newValue) {
        map.put(Thread.currentThread(), newValue);
    }

    public T get() {
        return map.get(Thread.currentThread());
    }

    public void remove() {
        map.remove(Thread.currentThread());
    }
}
```

148、解释一下什么叫 AOP（面向切面编程）？
答：AOP（Aspect-Oriented Programming）指一种程序设计范型，该范型以一种称为切面（aspect）的语言构造为基础，切面是一种新的模块化机制，用来描述分散在对象、类或方法中的横切关注点（crosscutting concern）。

149、你是如何理解"横切关注"这个概念的？
答："横切关注"是会影响到整个应用程序的关注功能，它跟正常的业务逻辑是正交的，没有必然的联系，但是几乎所有的业务逻辑都会涉及到这些关注功能。通常，事务、日志、安全性等关注就是应用中的横切关注功能。

150、你如何理解 AOP 中的连接点（Joinpoint）、切点（Pointcut）、增强（Advice）、引

介（Introduction）、织入（Weaving）、切面（Aspect）这些概念？
答：
a. 连接点（Joinpoint）：程序执行的某个特定位置（如：某个方法调用前、调用后，方法抛出异常后）。一个类或一段程序代码拥有一些具有边界性质的特定点，这些代码中的特定点就是连接点。Spring 仅支持方法的连接点。
b. 切点（Pointcut）：如果连接点相当于数据中的记录，那么切点相当于查询条件，一个切点可以匹配多个连接点。Spring AOP 的规则解析引擎负责解析切点所设定的查询条件，找到对应的连接点。
c. 增强（Advice）：增强是织入到目标类连接点上的一段程序代码。Spring 提供的增强接口都是带方位名的，如：BeforeAdvice、AfterReturningAdvice、ThrowsAdvice 等。很多资料上将增强译为“通知”，这明显是个词不达意的翻译，让很多程序员困惑了许久。

说明： Advice 在国内的很多书面资料中都被翻译成"通知"，但是很显然这个翻译无法表达其本质，有少量的读物上将这个词翻译为"增强"，这个翻译是对 Advice 较为准确的诠释，我们通过 AOP 将横切关注功能加到原有的业务逻辑上，这就是对原有业务逻辑的一种增强，这种增强可以是前置增强、后置增强、返回后增强、抛异常时增强和包围型增强。

d. 引介（Introduction）：引介是一种特殊的增强，它为类添加一些属性和方法。这样，即使一个业务类原本没有实现某个接口，通过引介功能，可以动态的未该业务类添加接口的实现逻辑，让业务类成为这个接口的实现类。
e. 织入（Weaving）：织入是将增强添加到目标类具体连接点上的过程，AOP 有三种织入方式：①编译期织入：需要特殊的 Java 编译期（例如 AspectJ 的 ajc）；②装载期织入：要求使用特殊的类加载器，在装载类的时候对类进行增强；③运行时织入：在运行时为目标类生成代理实现增强。Spring 采用了动态代理的方式实现了运行时织入，而 AspectJ 采用了编译期织入和装载期织入的方式。
f. 切面（Aspect）：切面是由切点和增强（引介）组成的，它包括了对横切关注功能的定义，也包括了对连接点的定义。

补充：代理模式是 GoF 提出的 23 种设计模式中最为经典的模式之一，代理模式是对象的结构模式，它给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。简单的说，代理对象可以完成比原对象更多的职责，当需要为原对象添加横切关注功能时，就可以使用原对象的代理对象。我们在打开 Office 系列的 Word 文档时，如果文档中有插图，当文档刚加载时，文档中的插图都只是一个虚框占位符，等用户真正翻到某页要查看该图片时，才会真正加载这张图，这其实就是对代理模式的使用，代替真正图片的虚框就是一个虚拟代理；Hibernate 的 load 方法也是返回一个虚拟代理对象，等用户真正需要访问对象的属性时，才向数据库发出 SQL 语句获得真实对象。

下面用一个找枪手代考的例子演示代理模式的使用：

```
/**
 * 参考人员接口
 * @author 骆昊
 *
 */
public interface Candidate {

    /**
     * 答题
     */
    public void answerTheQuestions();
}
```

```
/**
 * 懒学生
 * @author 骆昊
 *
 */
public class LazyStudent implements Candidate {
    private String name;       // 姓名

    public LazyStudent(String name) {
        this.name = name;
    }

    @Override
    public void answerTheQuestions() {
        // 懒学生只能写出自己的名字不会答题
        System.out.println("姓名: " + name);
    }

}


/**
 * 枪手
 * @author 骆昊
 *
 */
public class Gunman implements Candidate {
    private Candidate target;   // 被代理对象

    public Gunman(Candidate target) {
        this.target = target;
    }

    @Override
    public void answerTheQuestions() {
        // 枪手要写上代考的学生的姓名
        target.answerTheQuestions();
        // 枪手要帮助懒学生答题并交卷
        System.out.println("奋笔疾书正确答案");
        System.out.println("交卷");
    }

}

public class ProxyTest1 {

    public static void main(String[] args) {
        Candidate c = new Gunman(new LazyStudent("王小二"));
        c.answerTheQuestions();
    }
}
```

说明：从 JDK 1.3 开始，Java 提供了动态代理技术，允许开发者在运行时创建接口的代

理实例，主要包括 Proxy 类和 InvocationHandler 接口。下面的例子使用动态代理为 ArrayList 编写一个代理，在添加和删除元素时，在控制台打印添加或删除的元素以及 ArrayList 的大小：

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.util.List;

public class ListProxy<T> implements InvocationHandler {
   private List<T> target;

   public ListProxy(List<T> target) {
      this.target = target;
   }

   @Override
   public Object invoke(Object proxy, Method method, Object[] args)
         throws Throwable {
      Object retVal = null;
      System.out.println("[" + method.getName() + ": " + args[0] + "]");
      retVal = method.invoke(target, args);
      System.out.println("[size=" + target.size() + "]");
      return retVal;
   }

}

import java.lang.reflect.Proxy;
import java.util.ArrayList;
import java.util.List;

public class ProxyTest2 {

   @SuppressWarnings("unchecked")
   public static void main(String[] args) {
      List<String> list = new ArrayList<String>();
      Class<?> clazz = list.getClass();
      ListProxy<String> myProxy = new ListProxy<String>(list);
      List<String> newList = (List<String>)
            Proxy.newProxyInstance(clazz.getClassLoader(),
            clazz.getInterfaces(), myProxy);
      newList.add("apple");
      newList.add("banana");
      newList.add("orange");
      newList.remove("banana");
   }
}
```

说明：使用 Java 的动态代理有一个局限性就是代理的类必须要实现接口，虽然面向接口编程是每个优秀的 Java 程序都知道的规则，但现实往往不尽如人意，对于没有实现接口的类如何为其生成代理呢？继承！继承是最经典的扩展已有代码能力的手段，虽然继承常常被初学者滥用，但继承也常常被进阶的程序员忽视。CGLib 采用非常底层的字节码生成技术，通过为一个类创建子类来生成代理，它弥补了 Java 动态代理的不足，因此 Spring 中动态代理和 CGLib 都是创建代理的重要手段，对于实现了接口的类就用动态代理为其生成代理类，而没有实现接口的类就用 CGLib 通过继承的方式为其创建代理。

151、Spring 中自动装配的方式有哪些？
答：
- no：不进行自动装配，手动设置 Bean 的依赖关系。
- byName：根据 Bean 的名字进行自动装配。
- byType：根据 Bean 的类型进行自动装配。
- constructor：类似于 byType，不过是应用于构造器的参数，如果正好有一个 Bean 与构造器的参数类型相同则可以自动装配，否则会导致错误。
- autodetect：如果有默认的构造器，则通过 constructor 的方式进行自动装配，否则使用 byType 的方式进行自动装配。

说明：自动装配没有自定义装配方式那么精确，而且不能自动装配简单属性（基本类型、字符串等），在使用时应注意。

152、Spring 中如何使用注解来配置 Bean？有哪些相关的注解？
答：首先需要在 Spring 配置文件中增加如下配置：

```
<context:component-scan base-package="org.example"/>
```

然后可以用@Component、@Controller、@Service、@Repository 注解来标注需要由 Spring IoC 容器进行对象托管的类。这几个注解没有本质区别，只不过@Controller 通常用于控制器，@Service 通常用于业务逻辑类，@Repository 通常用于仓储类（例如我们的 DAO 实现类），普通的类用@Component 来标注。

153、Spring 支持的事务管理类型有哪些？你在项目中使用哪种方式？
答：Spring 支持编程式事务管理和声明式事务管理。许多 Spring 框架的用户选择声明式事务管理，因为这种方式和应用程序的关联较少，因此更加符合轻量级容器的概念。声明式事务管理要优于编程式事务管理，尽管在灵活性方面它弱于编程式事务管理，因为编程式事务允许你通过代码控制业务。

事务分为全局事务和局部事务。全局事务由应用服务器管理，需要底层服务器 JTA 支持（如 WebLogic、WildFly 等）。局部事务和底层采用的持久化方案有关，例如使用 JDBC 进行持久化时，需要使用 Connetion 对象来操作事务；而采用 Hibernate 进行持久化时，需要使用 Session 对象来操作事务。

Spring 提供了如下所示的事务管理器。

事务管理器实现类	目标对象
DataSourceTransactionManager	注入 DataSource
HibernateTransactionManager	注入 SessionFactory
JdoTransactionManager	管理 JDO 事务
JtaTransactionManager	使用 JTA 管理事务
PersistenceBrokerTransactionManager	管理 Apache 的 OJB 事务

这些事务的父接口都是 PlatformTransactionManager。Spring 的事务管理机制是一种典型的策略模式，PlatformTransactionManager 代表事务管理接口，该接口定义了三个方法，该接口并不知道底层如何管理事务，但是它的实现类必须提供 getTransaction()方法（开启事务）、commit()方法（提交事务）、rollback()方法（回滚事务）的多态实现，这样就可以用不同的实现类代表不同的事务管理策略。使用 JTA 全局事务策略时，需要底层应用服务器支持，而不同的应用服务器所提供的 JTA 全局事务可能存在细节上的差异，因此实际配置全局事务管理器是可能需要使用 JtaTransactionManager 的子类，如：

WebLogicJtaTransactionManager（Oracle 的 WebLogic 服务器提供）、
UowJtaTransactionManager（IBM 的 WebSphere 服务器提供）等。

编程式事务管理如下所示。

```
<?xml version="1.0" encoding="UTF-8"?>
 <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"  xmlns:p="http://www.springframework.org/schema/p"
   xmlns:p="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <context:component-scan base-package="com.jackfrued"/>

    <bean id="propertyConfig"
       class="org.springframework.beans.factory.config.
  PropertyPlaceholderConfigurer">
       <property name="location">
          <value>jdbc.properties</value>
       </property>
    </bean>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
       <property name="driverClassName">
          <value>${db.driver}</value>
       </property>
       <property name="url">
          <value>${db.url}</value>
       </property>
       <property name="username">
          <value>${db.username}</value>
       </property>
       <property name="password">
          <value>${db.password}</value>
       </property>
    </bean>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
       <property name="dataSource">
          <ref bean="dataSource" />
       </property>
    </bean>

    <!-- JDBC 事务管理器 -->
    <bean id="transactionManager"
       class="org.springframework.jdbc.datasource.
     DataSourceTransactionManager"    scope="singleton">
       <property name="dataSource">
          <ref bean="dataSource" />
       </property>
    </bean>

    <!-- 声明事务模板 -->
    <bean id="transactionTemplate"
       class="org.springframework.transaction.support.
```

```
  TransactionTemplate">
      <property name="transactionManager">
        <ref bean="transactionManager" />
      </property>
    </bean>

</beans>

package com.jackfrued.dao.impl;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;

import com.jackfrued.dao.EmpDao;
import com.jackfrued.entity.Emp;

@Repository
public class EmpDaoImpl implements EmpDao {
   @Autowired
   private JdbcTemplate jdbcTemplate;

   @Override
   public boolean save(Emp emp) {
      String sql = "insert into emp values (?,?,?)";
      return jdbcTemplate.update(sql, emp.getId(), emp.getName(), emp.getBirthday()) == 1;
   }

}

package com.jackfrued.biz.impl;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.TransactionCallbackWithoutResult;
import org.springframework.transaction.support.TransactionTemplate;

import com.jackfrued.biz.EmpService;
import com.jackfrued.dao.EmpDao;
import com.jackfrued.entity.Emp;

@Service
public class EmpServiceImpl implements EmpService {
   @Autowired
   private TransactionTemplate txTemplate;
   @Autowired
   private EmpDao empDao;

   @Override
   public void addEmp(final Emp emp) {
      txTemplate.execute(new TransactionCallbackWithoutResult() {

         @Override
         protected void doInTransactionWithoutResult(TransactionStatus txStatus) {
            empDao.save(emp);
         }
      });
```

```
    }
}
```

声明式事务如下图所示，以 Spring 整合 Hibernate 3 为例，包括完整的 DAO 和业务逻辑代码。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.2.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.2.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.2.xsd">

    <!-- 配置由 Spring IoC 容器托管的对象对应的被注解的类所在的包 -->
    <context:component-scan base-package="com.jackfrued" />

    <!-- 配置通过自动生成代理实现 AOP 功能 -->
    <aop:aspectj-autoproxy />

    <!-- 配置数据库连接池 (DBCP) -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
        <!-- 配置驱动程序类 -->
        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <!-- 配置连接数据库的 URL -->
        <property name="url" value="jdbc:mysql://localhost:3306/myweb" />
        <!-- 配置访问数据库的用户名 -->
        <property name="username" value="root" />
        <!-- 配置访问数据库的口令 -->
        <property name="password" value="123456" />
        <!-- 配置最大连接数 -->
        <property name="maxActive" value="150" />
        <!-- 配置最小空闲连接数 -->
        <property name="minIdle" value="5" />
        <!-- 配置最大空闲连接数 -->
        <property name="maxIdle" value="20" />
        <!-- 配置初始连接数 -->
        <property name="initialSize" value="10" />
        <!-- 配置连接被泄露时是否生成日志 -->
        <property name="logAbandoned" value="true" />
        <!-- 配置是否删除超时连接 -->
        <property name="removeAbandoned" value="true" />
        <!-- 配置删除超时连接的超时门限值(以秒为单位) -->
        <property name="removeAbandonedTimeout" value="120" />
        <!-- 配置超时等待时间(以毫秒为单位) -->
        <property name="maxWait" value="5000" />
```

```
        <!-- 配置空闲连接回收器线程运行的时间间隔(以毫秒为单位) -->
        <property name="timeBetweenEvictionRunsMillis" value="300000" />
        <!-- 配置连接空闲多长时间后(以毫秒为单位)被断开连接 -->
        <property name="minEvictableIdleTimeMillis" value="60000" />
    </bean>

    <!-- 配置 Spring 提供的支持注解 ORM 映射的 Hibernate 会话工厂 -->
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
        <!-- 通过 setter 注入数据源属性 -->
        <property name="dataSource" ref="dataSource" />
        <!-- 配置实体类所在的包 -->
        <property name="packagesToScan" value="com.jackfrued.entity" />
        <!-- 配置 Hibernate 的相关属性 -->
        <property name="hibernateProperties">
            <!-- 在项目调试完成后要删除 show_sql 和 format_sql 属性否则对性能有显著影响 -->
            <value>
                hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
            </value>
        </property>
    </bean>

    <!-- 配置 Spring 提供的 Hibernate 事务管理器 -->
    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <!-- 通过 setter 注入 Hibernate 会话工厂 -->
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <!-- 配置基于注解配置声明式事务 -->
    <tx:annotation-driven />

</beans>

package com.jackfrued.dao;

import java.io.Serializable;
import java.util.List;

import com.jackfrued.comm.QueryBean;
import com.jackfrued.comm.QueryResult;

/**
 * 数据访问对象接口(以对象为单位封装 CRUD 操作)
 * @author 骆昊
 *
 * @param <E> 实体类型
 * @param <K> 实体标识字段的类型
 */
public interface BaseDao <E, K extends Serializable> {

    /**
     * 新增
     * @param entity 业务实体对象
     * @return 增加成功返回实体对象的标识
```

```
 */
public K save(E entity);

/**
 * 删除
 * @param entity 业务实体对象
 */
public void delete(E entity);

/**
 * 根据 ID 删除
 * @param id 业务实体对象的标识
 * @return 删除成功返回 true 否则返回 false
 */
public boolean deleteById(K id);

/**
 * 修改
 * @param entity 业务实体对象
 * @return 修改成功返回 true 否则返回 false
 */
public void update(E entity);

/**
 * 根据 ID 查找业务实体对象
 * @param id 业务实体对象的标识
 * @return 业务实体对象对象或 null
 */
public E findById(K id);

/**
 * 根据 ID 查找业务实体对象
 * @param id 业务实体对象的标识
 * @param lazy 是否使用延迟加载
 * @return 业务实体对象对象
 */
public E findById(K id, boolean lazy);

/**
 * 查找所有业务实体对象
 * @return 装所有业务实体对象的列表容器
 */
public List<E> findAll();

/**
 * 分页查找业务实体对象
 * @param page 页码
 * @param size 页面大小
 * @return 查询结果对象
 */
public QueryResult<E> findByPage(int page, int size);

/**
 * 分页查找业务实体对象
```

```
     * @param queryBean 查询条件对象
     * @param page 页码
     * @param size 页面大小
     * @return 查询结果对象
     */
    public QueryResult<E> findByPage(QueryBean queryBean, int page, int size);

}

package com.jackfrued.dao;

import java.io.Serializable;
import java.util.List;

import com.jackfrued.comm.QueryBean;
import com.jackfrued.comm.QueryResult;

/**
 * BaseDao 的缺省适配器
 * @author 骆昊
 *
 * @param <E> 实体类型
 * @param <K> 实体标识字段的类型
 */
public abstract class BaseDaoAdapter<E, K extends Serializable> implements
        BaseDao<E, K> {

    @Override
    public K save(E entity) {
        return null;
    }

    @Override
    public void delete(E entity) {
    }

    @Override
    public boolean deleteById(K id) {
        E entity = findById(id);
        if(entity != null) {
            delete(entity);
            return true;
        }
        return false;
    }

    @Override
    public void update(E entity) {
    }

    @Override
    public E findById(K id) {
        return null;
    }

    @Override
```

```
    public E findById(K id, boolean lazy) {
        return null;
    }

    @Override
    public List<E> findAll() {
        return null;
    }

    @Override
    public QueryResult<E> findByPage(int page, int size) {
        return null;
    }

    @Override
    public QueryResult<E> findByPage(QueryBean queryBean, int page, int size) {
        return null;
    }

}

package com.jackfrued.dao;

import java.io.Serializable;
import java.lang.reflect.ParameterizedType;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;

import com.jackfrued.comm.HQLQueryBean;
import com.jackfrued.comm.QueryBean;
import com.jackfrued.comm.QueryResult;

/**
 * 基于 Hibernate 的 BaseDao 实现类
 * @author 骆昊
 *
 * @param <E> 实体类型
 * @param <K> 主键类型
 */
@SuppressWarnings(value = {"unchecked"})
public abstract class BaseDaoHibernateImpl<E, K extends Serializable> extends
BaseDaoAdapter<E, K> {
    @Autowired
    protected SessionFactory sessionFactory;

    private Class<?> entityClass;      // 业务实体的类对象
    private String entityName;         // 业务实体的名字

    public BaseDaoHibernateImpl() {
        ParameterizedType pt = (ParameterizedType) this.getClass().getGenericSuperclass();
```

```
        entityClass = (Class<?>) pt.getActualTypeArguments()[0];
        entityName = entityClass.getSimpleName();
    }

    @Override
    public K save(E entity) {
        return (K) sessionFactory.getCurrentSession().save(entity);
    }

    @Override
    public void delete(E entity) {
        sessionFactory.getCurrentSession().delete(entity);
    }

    @Override
    public void update(E entity) {
        sessionFactory.getCurrentSession().update(entity);
    }

    @Override
    public E findById(K id) {
        return findById(id, false);
    }

    @Override
    public E findById(K id, boolean lazy) {
        Session session = sessionFactory.getCurrentSession();
        return (E) (lazy? session.load(entityClass, id) : session.get(entityClass, id));
    }

    @Override
    public List<E> findAll() {
        return sessionFactory.getCurrentSession().createCriteria(entityClass).list();
    }

    @Override
    public QueryResult<E> findByPage(int page, int size) {
        return new QueryResult<E>(
            findByHQLAndPage("from " + entityName , page, size),
            getCountByHQL("select count(*) from " + entityName)
        );
    }

    @Override
    public QueryResult<E> findByPage(QueryBean queryBean, int page, int size) {
        if(queryBean instanceof HQLQueryBean) {
            HQLQueryBean hqlQueryBean = (HQLQueryBean) queryBean;
            return new QueryResult<E>(
                findByHQLAndPage(hqlQueryBean.getQueryString(), page, size,
hqlQueryBean.getParameters()),
                getCountByHQL(hqlQueryBean.getCountString(), hqlQueryBean.getParameters())
            );
        }
        return null;
    }

    /**
```

```
 * 根据 HQL 和可变参数列表进行查询
 * @param hql 基于 HQL 的查询语句
 * @param params 可变参数列表
 * @return 持有查询结果的列表容器或空列表容器
 */
protected List<E> findByHQL(String hql, Object... params) {
   return this.findByHQL(hql, getParamList(params));
}

/**
 * 根据 HQL 和参数列表进行查询
 * @param hql 基于 HQL 的查询语句
 * @param params 查询参数列表
 * @return 持有查询结果的列表容器或空列表容器
 */
protected List<E> findByHQL(String hql, List<Object> params) {
   List<E> list = createQuery(hql, params).list();
   return list != null && list.size() > 0 ? list : Collections.EMPTY_LIST;
}

/**
 * 根据 HQL 和参数列表进行分页查询
 * @param hql 基于 HQL 的查询语句
 * @page 页码
 * @size 页面大小
 * @param params 可变参数列表
 * @return 持有查询结果的列表容器或空列表容器
 */
protected List<E> findByHQLAndPage(String hql, int page, int size, Object... params) {
   return this.findByHQLAndPage(hql, page, size, getParamList(params));
}

/**
 * 根据 HQL 和参数列表进行分页查询
 * @param hql 基于 HQL 的查询语句
 * @page 页码
 * @size 页面大小
 * @param params 查询参数列表
 * @return 持有查询结果的列表容器或空列表容器
 */
protected List<E> findByHQLAndPage(String hql, int page, int size, List<Object> params) {
   List<E> list = createQuery(hql, params)
         .setFirstResult((page - 1) * size)
         .setMaxResults(size)
         .list();
   return list != null && list.size() > 0 ? list : Collections.EMPTY_LIST;
}

/**
 * 查询满足条件的记录数
 * @param hql 基于 HQL 的查询语句
 * @param params 可变参数列表
 * @return 满足查询条件的总记录数
 */
```

```
    protected long getCountByHQL(String hql, Object... params) {
        return this.getCountByHQL(hql, getParamList(params));
    }

    /**
     * 查询满足条件的记录数
     * @param hql 基于 HQL 的查询语句
     * @param params 参数列表容器
     * @return 满足查询条件的总记录数
     */
    protected long getCountByHQL(String hql, List<Object> params) {
        return (Long) createQuery(hql, params).uniqueResult();
    }

    // 创建 Hibernate 查询对象(Query)
    private Query createQuery(String hql, List<Object> params) {
        Query query = sessionFactory.getCurrentSession().createQuery(hql);
        for(int i = 0; i < params.size(); i++) {
            query.setParameter(i, params.get(i));
        }
        return query;
    }

    // 将可变参数列表组装成列表容器
    private List<Object> getParamList(Object... params) {
        List<Object> paramList = new ArrayList<>();
        if(params != null) {
            for(int i = 0; i < params.length; i++) {
                paramList.add(params[i]);
            }
        }
        return paramList.size() == 0? Collections.EMPTY_LIST : paramList;
    }

}

package com.jackfrued.comm;

import java.util.List;

/**
 * 查询条件的接口
 * @author 骆昊
 *
 */
public interface QueryBean {

    /**
     * 添加排序字段
     * @param fieldName 用于排序的字段
     * @param asc 升序还是降序
     * @return 查询条件对象自身(方便级联编程)
     */
    public QueryBean addOrder(String fieldName, boolean asc);

    /**
```

```
    * 添加排序字段
    * @param available 是否添加此排序字段
    * @param fieldName 用于排序的字段
    * @param asc 升序还是降序
    * @return 查询条件对象自身(方便级联编程)
    */
   public QueryBean addOrder(boolean available, String fieldName, boolean asc);

   /**
    * 添加查询条件
    * @param condition 条件
    * @param params 替换掉条件中参数占位符的参数
    * @return 查询条件对象自身(方便级联编程)
    */
   public QueryBean addCondition(String condition, Object... params);

   /**
    * 添加查询条件
    * @param available 是否需要添加此条件
    * @param condition 条件
    * @param params 替换掉条件中参数占位符的参数
    * @return 查询条件对象自身(方便级联编程)
    */
   public QueryBean addCondition(boolean available, String condition, Object... params);

   /**
    * 获得查询语句
    * @return 查询语句
    */
   public String getQueryString();

   /**
    * 获取查询记录数的查询语句
    * @return 查询记录数的查询语句
    */
   public String getCountString();

   /**
    * 获得查询参数
    * @return 查询参数的列表容器
    */
   public List<Object> getParameters();
}

package com.jackfrued.comm;

import java.util.List;

/**
 * 查询结果
 * @author 骆昊
 *
 * @param <T> 泛型参数
 */
```

```
public class QueryResult<T> {
    private List<T> result;    // 持有查询结果的列表容器
    private long totalRecords;  // 查询到的总记录数

    /**
     * 构造器
     */
    public QueryResult() {
    }

    /**
     * 构造器
     * @param result 持有查询结果的列表容器
     * @param totalRecords 查询到的总记录数
     */
    public QueryResult(List<T> result, long totalRecords) {
        this.result = result;
        this.totalRecords = totalRecords;
    }

    public List<T> getResult() {
        return result;
    }

    public void setResult(List<T> result) {
        this.result = result;
    }

    public long getTotalRecords() {
        return totalRecords;
    }

    public void setTotalRecords(long totalRecords) {
        this.totalRecords = totalRecords;
    }
}

package com.jackfrued.dao;

import com.jackfrued.comm.QueryResult;
import com.jackfrued.entity.Dept;

/**
 * 部门数据访问对象接口
 * @author 骆昊
 *
 */
public interface DeptDao extends BaseDao<Dept, Integer> {

    /**
     * 分页查询顶级部门
     * @param page 页码
     * @param size 页码大小
     * @return 查询结果对象
     */
    public QueryResult<Dept> findTopDeptByPage(int page, int size);
```

```
}

package com.jackfrued.dao.impl;

import java.util.List;

import org.springframework.stereotype.Repository;

import com.jackfrued.comm.QueryResult;
import com.jackfrued.dao.BaseDaoHibernateImpl;
import com.jackfrued.dao.DeptDao;
import com.jackfrued.entity.Dept;

@Repository
public class DeptDaoImpl extends BaseDaoHibernateImpl<Dept, Integer> implements DeptDao {
   private static final String HQL_FIND_TOP_DEPT = " from Dept as d where d.superiorDept is
null ";

   @Override
   public QueryResult<Dept> findTopDeptByPage(int page, int size) {
      List<Dept> list = findByHQLAndPage(HQL_FIND_TOP_DEPT, page, size);
      long totalRecords = getCountByHQL(" select count(*) " + HQL_FIND_TOP_DEPT);
      return new QueryResult<>(list, totalRecords);
   }

}

package com.jackfrued.comm;

import java.util.List;

/**
 * 分页器
 * @author 骆昊
 *
 * @param <T> 分页数据对象的类型
 */
public class PageBean<T> {
   private static final int DEFAUL_INIT_PAGE = 1;
   private static final int DEFAULT_PAGE_SIZE = 10;
   private static final int DEFAULT_PAGE_COUNT = 5;

   private List<T> data;          // 分页数据
   private PageRange pageRange;   // 页码范围
   private int totalPage;         // 总页数
   private int size;              // 页面大小
   private int currentPage;       // 当前页码
   private int pageCount;         // 页码数量

   /**
    * 构造器
    * @param currentPage 当前页码
    * @param size 页码大小
    * @param pageCount 页码数量
    */
```

```
public PageBean(int currentPage, int size, int pageCount) {
    this.currentPage = currentPage > 0 ? currentPage : 1;
    this.size = size > 0 ? size : DEFAULT_PAGE_SIZE;
    this.pageCount = pageCount > 0 ? size : DEFAULT_PAGE_COUNT;
}

/**
 * 构造器
 * @param currentPage 当前页码
 * @param size 页码大小
 */
public PageBean(int currentPage, int size) {
    this(currentPage, size, DEFAULT_PAGE_COUNT);
}

/**
 * 构造器
 * @param currentPage 当前页码
 */
public PageBean(int currentPage) {
    this(currentPage, DEFAULT_PAGE_SIZE, DEFAULT_PAGE_COUNT);
}

/**
 * 构造器
 */
public PageBean() {
    this(DEFAUL_INIT_PAGE, DEFAULT_PAGE_SIZE, DEFAULT_PAGE_COUNT);
}

public List<T> getData() {
    return data;
}

public int getStartPage() {
    return pageRange != null ? pageRange.getStartPage() : 1;
}

public int getEndPage() {
    return pageRange != null ? pageRange.getEndPage() : 1;
}

public long getTotalPage() {
    return totalPage;
}

public int getSize() {
    return size;
}

public int getCurrentPage() {
    return currentPage;
}

/**
 * 将查询结果转换为分页数据
```

```
     * @param queryResult 查询结果对象
     */
    public void transferQueryResult(QueryResult<T> queryResult) {
        long totalRecords = queryResult.getTotalRecords();

        data = queryResult.getResult();
        totalPage = (int) ((totalRecords + size - 1) / size);
        totalPage = totalPage >= 0 ? totalPage : Integer.MAX_VALUE;
        this.pageRange = new PageRange(pageCount, currentPage, totalPage);
    }

}

package com.jackfrued.comm;

/**
 * 页码范围
 * @author 骆昊
 *
 */
public class PageRange {
    private int startPage;  // 起始页码
    private int endPage;    // 终止页码

    /**
     * 构造器
     * @param pageCount 总共显示几个页码
     * @param currentPage 当前页码
     * @param totalPage 总页数
     */
    public PageRange(int pageCount, int currentPage, int totalPage) {
        startPage = currentPage - (pageCount - 1) / 2;
        endPage = currentPage + pageCount / 2;
        if(startPage < 1) {
            startPage = 1;
            endPage = totalPage > pageCount ? pageCount : totalPage;
        }
        if (endPage > totalPage) {
            endPage = totalPage;
            startPage = (endPage - pageCount > 0) ? endPage - pageCount + 1 : 1;
        }
    }

    /**
     * 获得起始页页码
     * @return 起始页页码
     */
    public int getStartPage() {
        return startPage;
    }

    /**
     * 获得终止页页码
     * @return 终止页页码
     */
    public int getEndPage() {
```

```
        return endPage;
    }

}

package com.jackfrued.biz;

import com.jackfrued.comm.PageBean;
import com.jackfrued.entity.Dept;

/**
 * 部门业务逻辑接口
 * @author 骆昊
 *
 */
public interface DeptService {

    /**
     * 创建新的部门
     * @param department 部门对象
     * @return 创建成功返回 true 否则返回 false
     */
    public boolean createNewDepartment(Dept department);

    /**
     * 删除指定部门
     * @param id 要删除的部门的编号
     * @return 删除成功返回 true 否则返回 false
     */
    public boolean deleteDepartment(Integer id);

    /**
     * 分页获取顶级部门
     * @param page 页码
     * @param size 页码大小
     * @return 部门对象的分页器对象
     */
    public PageBean<Dept> getTopDeptByPage(int page, int size);

}

package com.jackfrued.biz.impl;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.jackfrued.biz.DeptService;
import com.jackfrued.comm.PageBean;
import com.jackfrued.comm.QueryResult;
import com.jackfrued.dao.DeptDao;
import com.jackfrued.entity.Dept;

@Service
@Transactional  // 声明式事务的注解
public class DeptServiceImpl implements DeptService {
```

```
    @Autowired
    private DeptDao deptDao;

    @Override
    public boolean createNewDepartment(Dept department) {
        return deptDao.save(department) != null;
    }

    @Override
    public boolean deleteDepartment(Integer id) {
        return deptDao.deleteById(id);
    }

    @Override
    public PageBean<Dept> getTopDeptByPage(int page, int size) {
        QueryResult<Dept> queryResult = deptDao.findTopDeptByPage(page, size);
        PageBean<Dept> pageBean = new PageBean<>(page, size);
        pageBean.transferQueryResult(queryResult);
        return pageBean;
    }
}
```

154、如何在 Web 项目中配置 Spring 的 IoC 容器？
答：如果需要在 Web 项目中使用 Spring 的 IoC 容器，可以在 Web 项目配置文件 web.xml 中做出如下配置：

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</context-param>

<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

155、如何在 Web 项目中配置 Spring MVC？
答：要使用 Spring MVC 需要在 Web 项目配置文件中配置其前端控制器 DispatcherServlet，如下所示：

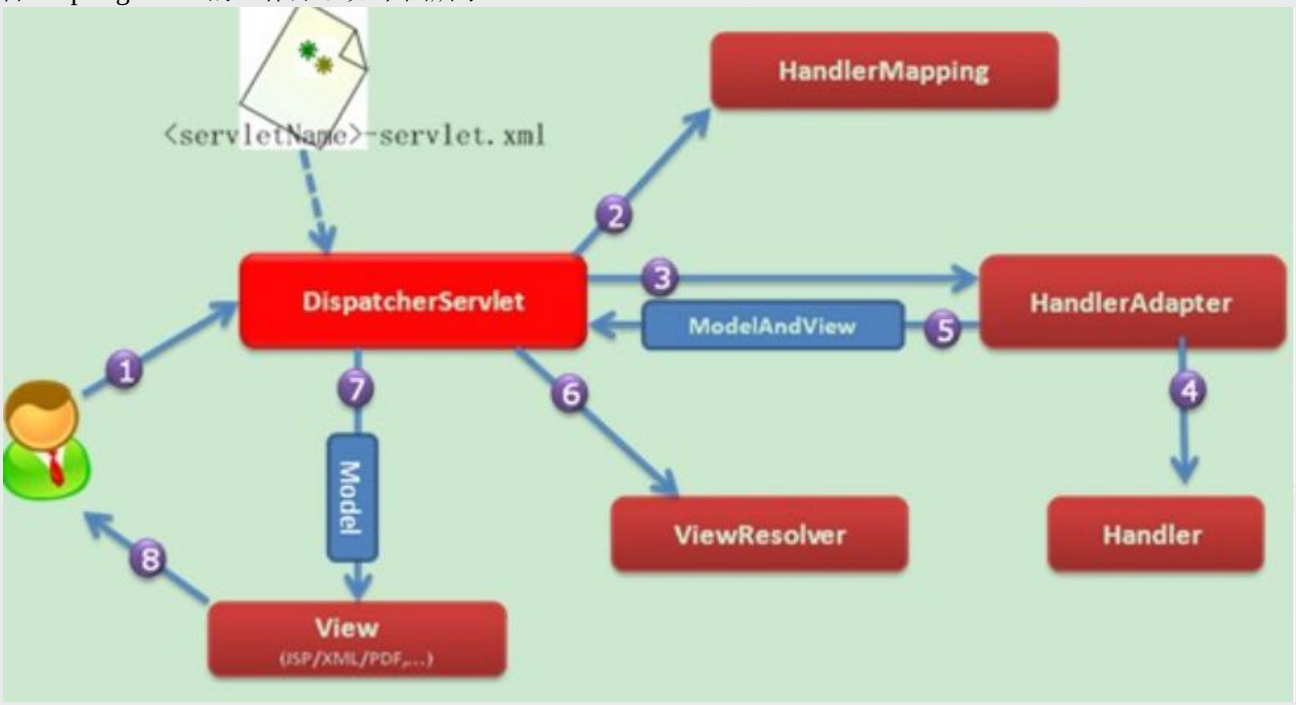
说明：上面的配置中使用了*.html 的后缀映射，这样做一方面不能够通过 URL 推断采用了何种服务器端的技术，另一方面可以欺骗搜索引擎，因为搜索引擎不会搜索动态页面，这种做法称为伪静态化。

```
    <web-app>

        <servlet>
            <servlet-name>example</servlet-name>
            <servlet-class>
                org.springframework.web.servlet.DispatcherServlet
            </servlet-class>
            <load-on-startup>1</load-on-startup>
        </servlet>

        <servlet-mapping>
```

```
        <servlet-name>example</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>

</web-app>
```

156、Spring MVC 的工作原理是怎样的？

答：Spring MVC 的工作原理如下图所示：



① 客户端的所有请求都交给前端控制器 DispatcherServlet 来处理，它会负责调用系统的其他模块来真正处理用户的请求。

② DispatcherServlet 收到请求后，将根据请求的信息（包括 URL、HTTP 协议方法、请求头、请求参数、Cookie 等）以及 HandlerMapping 的配置找到处理该请求的 Handler（任何一个对象都可以作为请求的 Handler）。

③在这个地方 Spring 会通过 HandlerAdapter 对该处理器进行封装。

④ HandlerAdapter 是一个适配器，它用统一的接口对各种 Handler 中的方法进行调用。

⑤ Handler 完成对用户请求的处理后，会返回一个 ModelAndView 对象给 DispatcherServlet，ModelAndView 顾名思义，包含了数据模型以及相应的视图的信息。

⑥ ModelAndView 的视图是逻辑视图，DispatcherServlet 还要借助 ViewResolver 完成从逻辑视图到真实视图对象的解析工作。

⑦ 当得到真正的视图对象后，DispatcherServlet 会利用视图对象对模型数据进行渲染。

⑧ 客户端得到响应，可能是一个普通的 HTML 页面，也可以是 XML 或 JSON 字符串，还可以是一张图片或者一个 PDF 文件。

157、如何在 Spring IoC 容器中配置数据源？

答：

DBCP 配置：

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
```

```
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

C3P0 配置：

```
<bean id="dataSource"
       class="com.mchange.v2.c3p0.ComboPooledDataSource" destroy-method="close">
    <property name="driverClass" value="${jdbc.driverClassName}"/>
    <property name="jdbcUrl" value="${jdbc.url}"/>
    <property name="user" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>

<context:property-placeholder location="jdbc.properties"/>
```

提示： DBCP 的详细配置在第 153 题中已经完整的展示过了。

158、如何配置配置事务增强？
答：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:aop="http://www.springframework.org/schema/aop"
     xmlns:tx="http://www.springframework.org/schema/tx"
     xsi:schemaLocation="
     http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/tx
     http://www.springframework.org/schema/tx/spring-tx.xsd
     http://www.springframework.org/schema/aop
     http://www.springframework.org/schema/aop/spring-aop.xsd">

  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- the transactional advice -->
  <tx:advice id="txAdvice" transaction-manager="txManager">
  <!-- the transactional semantics... -->
  <tx:attributes>
    <!-- all methods starting with 'get' are read-only -->
    <tx:method name="get*" read-only="true"/>
    <!-- other methods use the default transaction settings (see below) -->
    <tx:method name="*"/>
  </tx:attributes>
  </tx:advice>

  <!-- ensure that the above transactional advice runs for any execution
    of an operation defined by the FooService interface -->
  <aop:config>
  <aop:pointcut id="fooServiceOperation"
    expression="execution(* x.y.service.FooService.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>
  </aop:config>
```

```
  <!-- don't forget the DataSource -->
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
  <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
  <property name="url" value="jdbc:oracle:thin:@localhost:1521:orcl"/>
  <property name="username" value="scott"/>
  <property name="password" value="tiger"/>
  </bean>

  <!-- similarly, don't forget the PlatformTransactionManager -->
  <bean id="txManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other <bean/> definitions here -->

</beans>
```

159、选择使用 Spring 框架的原因（Spring 框架为企业级开发带来的好处有哪些）？
答：可以从以下几个方面作答：
- 非侵入式：支持基于 POJO 的编程模式，不强制性的要求实现 Spring 框架中的接口或继承 Spring 框架中的类。
- IoC 容器：IoC 容器帮助应用程序管理对象以及对象之间的依赖关系，对象之间的依赖关系如果发生了改变只需要修改配置文件而不是修改代码，因为代码的修改可能意味着项目的重新构建和完整的回归测试。有了 IoC 容器，程序员再也不需要自己编写工厂、单例，这一点特别符合 Spring 的精神"不要重复的发明轮子"。
- AOP（面向切面编程）：将所有的横切关注功能封装到切面（aspect）中，通过配置的方式将横切关注功能动态添加到目标代码上，进一步实现了业务逻辑和系统服务之间的分离。另一方面，有了 AOP 程序员可以省去很多自己写代理类的工作。
- MVC：Spring 的 MVC 框架是非常优秀的，从各个方面都可以甩 Struts 2 几条街，为 Web 表示层提供了更好的解决方案。
- 事务管理：Spring 以宽广的胸怀接纳多种持久层技术，并且为其提供了声明式的事务管理，在不需要任何一行代码的情况下就能够完成事务管理。
- 其他：选择 Spring 框架的原因还远不止于此，Spring 为 Java 企业级开发提供了一站式选择，你可以在需要的时候使用它的部分和全部，更重要的是，你甚至可以在感觉不到 Spring 存在的情况下，在你的项目中使用 Spring 提供的各种优秀的功能。

160、Spring IoC 容器配置 Bean 的方式？
答：
- 基于 XML 文件进行配置。
- 基于注解进行配置。
- 基于 Java 程序进行配置（Spring 3+）

```
package com.jackfrued.bean;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class Person {
   private String name;
   private int age;
```

```
    @Autowired
    private Car car;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void setCar(Car car) {
        this.car = car;
    }

    @Override
    public String toString() {
        return "Person [name=" + name + ", age=" + age + ", car=" + car + "]";
    }

}

package com.jackfrued.bean;

import org.springframework.stereotype.Component;

@Component
public class Car {
    private String brand;
    private int maxSpeed;

    public Car(String brand, int maxSpeed) {
        this.brand = brand;
        this.maxSpeed = maxSpeed;
    }

    @Override
    public String toString() {
        return "Car [brand=" + brand + ", maxSpeed=" + maxSpeed + "]";
    }

}

package com.jackfrued.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.jackfrued.bean.Car;
import com.jackfrued.bean.Person;

@Configuration
public class AppConfig {

    @Bean
    public Car car() {
        return new Car("Benz", 320);
    }

    @Bean
```

```
    public Person person() {
        return new Person("骆昊", 34);
    }
}

package com.jackfrued.test;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.jackfrued.bean.Person;
import com.jackfrued.config.AppConfig;

class Test {

    public static void main(String[] args) {
        // TWR (Java 7+)
        try(ConfigurableApplicationContext factory = new
AnnotationConfigApplicationContext(AppConfig.class)) {
            Person person = factory.getBean(Person.class);
            System.out.println(person);
        }
    }
}
```

161、阐述 Spring 框架中 Bean 的生命周期？
答：
① Spring IoC 容器找到关于 Bean 的定义并实例化该 Bean。
② Spring IoC 容器对 Bean 进行依赖注入。
③ 如果 Bean 实现了 BeanNameAware 接口，则将该 Bean 的 id 传给 setBeanName 方法。
④ 如果 Bean 实现了 BeanFactoryAware 接口，则将 BeanFactory 对象传给 setBeanFactory 方法。
⑤ 如果 Bean 实现了 BeanPostProcessor 接口，则调用其 postProcessBeforeInitialization 方法。
⑥ 如果 Bean 实现了 InitializingBean 接口，则调用其 afterPropertySet 方法。
⑦ 如果有和 Bean 关联的 BeanPostProcessors 对象，则这些对象的 postProcessAfterInitialization 方法被调用。
⑧ 当销毁 Bean 实例时，如果 Bean 实现了 DisposableBean 接口，则调用其 destroy 方法。

162、依赖注入时如何注入集合属性？
答：可以在定义 Bean 属性时，通过<list> / <set> / <map> / <props>分别为其注入列表、集合、映射和键值都是字符串的映射属性。

163、Spring 中的自动装配有哪些限制？
答：
- 如果使用了构造器注入或者 setter 注入，那么将覆盖自动装配的依赖关系。
- 基本数据类型的值、字符串字面量、类字面量无法使用自动装配来注入。
- 优先考虑使用显式的装配来进行更精确的依赖注入而不是使用自动装配。

164、在 Web 项目中如何获得 Spring 的 IoC 容器？
答：

```
WebApplicationContext ctx =
WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

165. 大型网站在架构上应当考虑哪些问题？
答：
- 分层：分层是处理任何复杂系统最常见的手段之一，将系统横向切分成若干个层面，每个层面只承担单一的职责，然后通过下层为上层提供的基础设施和服务以及上层对下层的调用来形成一个完整的复杂的系统。计算机网络的开放系统互联参考模型（OSI/RM）和 Internet 的 TCP/IP 模型都是分层结构，大型网站的软件系统也可以使用分层的理念将其分为持久层（提供数据存储和访问服务）、业务层（处理业务逻辑，系统中最核心的部分）和表示层（系统交互、视图展示）。需要指出的是：（1）分层是逻辑上的划分，在物理上可以位于同一设备上也可以在不同的设备上部署不同的功能模块，这样可以使用更多的计算资源来应对用户的并发访问；（2）层与层之间应当有清晰的边界，这样分层才有意义，才更利于软件的开发和维护。
- 分割：分割是对软件的纵向切分。我们可以将大型网站的不同功能和服务分割开，形成高内聚低耦合的功能模块（单元）。在设计初期可以做一个粗粒度的分割，将网站分割为若干个功能模块，后期还可以进一步对每个模块进行细粒度的分割，这样一方面有助于软件的开发和维护，另一方面有助于分布式的部署，提供网站的并发处理能力和功能的扩展。
- 分布式：除了上面提到的内容，网站的静态资源（JavaScript、CSS、图片等）也可以采用独立分布式部署并采用独立的域名，这样可以减轻应用服务器的负载压力，也使得浏览器对资源的加载更快。数据的存取也应该是分布式的，传统的商业级关系型数据库产品基本上都支持分布式部署，而新生的 NoSQL 产品几乎都是分布式的。当然，网站后台的业务处理也要使用分布式技术，例如查询索引的构建、数据分析等，这些业务计算规模庞大，可以使用 Hadoop 以及 MapReduce 分布式计算框架来处理。
- 集群：集群使得有更多的服务器提供相同的服务，可以更好的提供对并发的支持。
- 缓存：所谓缓存就是用空间换取时间的技术，将数据尽可能放在距离计算最近的位置。使用缓存是网站优化的第一定律。我们通常说的 CDN、反向代理、热点数据都是对缓存技术的使用。
- 异步：异步是实现软件实体之间解耦合的又一重要手段。异步架构是典型的生产者消费者模式，二者之间没有直接的调用关系，只要保持数据结构不变，彼此功能实现可以随意变化而不互相影响，这对网站的扩展非常有利。使用异步处理还可以提高系统可用性，加快网站的响应速度（用 Ajax 加载数据就是一种异步技术），同时还可以起到削峰作用（应对瞬时高并发）。&quot；能推迟处理的都要推迟处理"是网站优化的第二定律，而异步是践行网站优化第二定律的重要手段。
- 冗余：各种服务器都要提供相应的冗余服务器以便在某台或某些服务器宕机时还能保证网站可以正常工作，同时也提供了灾难恢复的可能性。冗余是网站高可用性的重要保证。

166、你用过的网站前端优化的技术有哪些？
答：
① 浏览器访问优化：
- 减少 HTTP 请求数量：合并 CSS、合并 JavaScript、合并图片（CSS Sprite）
- 使用浏览器缓存：通过设置 HTTP 响应头中的 Cache-Control 和 Expires 属性，将 CSS、JavaScript、图片等在浏览器中缓存，当这些静态资源需要更新时，可以更新 HTML 文件中的引用来让浏览器重新请求新的资源
- 启用压缩
- CSS 前置，JavaScript 后置
- 减少 Cookie 传输
② CDN 加速：CDN（Content Distribute Network）的本质仍然是缓存，将数据缓存在离用户最近的地方，CDN 通常部署在网络运营商的机房，不仅可以提升响应速度，还可以减少应用服务器的压力。当然，CDN 缓存的通常都是静态资源。
③ 反向代理：反向代理相当于应用服务器的一个门面，可以保护网站的安全性，也可以实现负载均衡的功能，当然最重要的是它缓存了用户访问的热点资源，可以直接从反向代理将某些内容返回给用户浏览器。

167、你使用过的应用服务器优化技术有哪些？
答：
① 分布式缓存：缓存的本质就是内存中的哈希表，如果设计一个优质的哈希函数，那么理论上哈希表读写的渐近时间复杂度为 O(1)。缓存主要用来存放那些读写比很高、变化很少的数据，这样应用程序读取数据时先到缓存中读取，如果没有或者数据已经失效再去访问数据库或文件系统，并根据拟定的规则将数据写入缓存。对网站数据的访问也符合二八定律（Pareto 分布，幂律分布），即 80%的访问都集中在 20%的数据上，如果能够将这 20%的数据缓存起来，那么系统的性能将得到显著的改善。当然，使用缓存需要解决以下几个问题：
- 频繁修改的数据；
- 数据不一致与脏读；
- 缓存雪崩（可以采用分布式缓存服务器集群加以解决，memcached 是广泛采用的解决方案）；
- 缓存预热；
- 缓存穿透（恶意持续请求不存在的数据）。
② 异步操作：可以使用消息队列将调用异步化，通过异步处理将短时间高并发产生的事件消息存储在消息队列中，从而起到削峰作用。电商网站在进行促销活动时，可以将用户的订单请求存入消息队列，这样可以抵御大量的并发订单请求对系统和数据库的冲击。目前，绝大多数的电商网站即便不进行促销活动，订单系统都采用了消息队列来处理。
③ 使用集群。
④ 代码优化：
- 多线程：基于 Java 的 Web 开发基本上都通过多线程的方式响应用户的并发请求，使用多线程技术在编程上要解决线程安全问题，主要可以考虑以下几个方面：A. 将对象设计为无状态对象（这和面向对象的编程观点是矛盾的，在面向对象的世界中被视为不良设计），这样就不会存在并发访问时对象状态不一致的问题。B. 在方法内部创建对象，这样对象由进入方法的线程创建，不会出现多个线程访问同一对象的问题。使用 ThreadLocal 将对象与线程绑定也是很好的做法，这一点在前面已经探讨过了。C. 对资源进行并发访问时应当使用合理的锁机制。
- 非阻塞 I/O： 使用单线程和非阻塞 I/O 是目前公认的比多线程的方式更能充分发挥服务器性能的应用模式，基于 Node.js 构建的服务器就采用了这样的方式。Java 在 JDK 1.4 中就引入了 NIO（Non-blocking I/O）,在 Servlet 3 规范中又引入了异步 Servlet 的概念，这些都为在服务器端采用非阻塞 I/O 提供了必要的基础。
- 资源复用：资源复用主要有两种方式，一是单例，二是对象池，我们使用的数据库连接池、线程池都是对象池化技术，这是典型的用空间换取时间的策略，另一方面也实现对资源的复用，从而避免了不必要的创建和释放资源所带来的开销。

168、什么是 XSS 攻击？什么是 SQL 注入攻击？什么是 CSRF 攻击？
答：
- XSS（Cross Site Script，跨站脚本攻击）是向网页中注入恶意脚本在用户浏览网页时在用户浏览器中执行恶意脚本的攻击方式。跨站脚本攻击分有两种形式：反射型攻击（诱使用户点击一个嵌入恶意脚本的链接以达到攻击的目标，目前有很多攻击者利用论坛、微博发布含有恶意脚本的 URL 就属于这种方式）和持久型攻击（将恶意脚本提交到被攻击网站的数据库中，用户浏览网页时，恶意脚本从数据库中被加载到页面执行，QQ 邮箱的早期版本就曾经被利用作为持久型跨站脚本攻击的平台）。XSS 虽然不是什么新鲜玩意，但是攻击的手法却不断翻新，防范 XSS 主要有两方面：消毒（对危险字符进行转义）和 HttpOnly（防范 XSS 攻击者窃取 Cookie 数据）。
- SQL 注入攻击是注入攻击最常见的形式（此外还有 OS 注入攻击（Struts 2 的高危漏洞就是通过 OGNL 实施 OS 注入攻击导致的）），当服务器使用请求参数构造 SQL 语句时，恶意的 SQL 被嵌入到 SQL 中交给数据库执行。SQL 注入攻击需要攻击者对数据库结构有所了解才能进行，攻击者想要获得表结构有多种方式：（1）如果使用开源系统搭建网站，数据库结构也是公开的（目前有很多现成的系统可以直接搭建论坛，电商网站，虽然方便快

捷但是风险是必须要认真评估的）；（2）错误回显（如果将服务器的错误信息直接显示在页面上，攻击者可以通过非法参数引发页面错误从而通过错误信息了解数据库结构，Web 应用应当设置友好的错误页，一方面符合最小惊讶原则，一方面屏蔽掉可能给系统带来危险的错误回显信息）；（3）盲注。防范 SQL 注入攻击也可以采用消毒的方式，通过正则表达式对请求参数进行验证，此外，参数绑定也是很好的手段，这样恶意的 SQL 会被当做 SQL 的参数而不是命令被执行，JDBC 中的 PreparedStatement 就是支持参数绑定的语句对象，从性能和安全性上都明显优于 Statement。

- CSRF 攻击（Cross Site Request Forgery，跨站请求伪造）是攻击者通过跨站请求，以合法的用户身份进行非法操作（如转账或发帖等）。CSRF 的原理是利用浏览器的 Cookie 或服务器的 Session，盗取用户身份，其原理如下图所示。防范 CSRF 的主要手段是识别请求者的身份，主要有以下几种方式：（1）在表单中添加令牌（token）；（2）验证码；（3）检查请求头中的 Referer（前面提到防图片盗链接也是用的这种方式）。令牌和验证都具有一次消费性的特征，因此在原理上一致的，但是验证码是一种糟糕的用户体验，不是必要的情况下不要轻易使用验证码，目前很多网站的做法是如果在短时间内多次提交一个表单未获得成功后才要求提供验证码，这样会获得较好的用户体验。

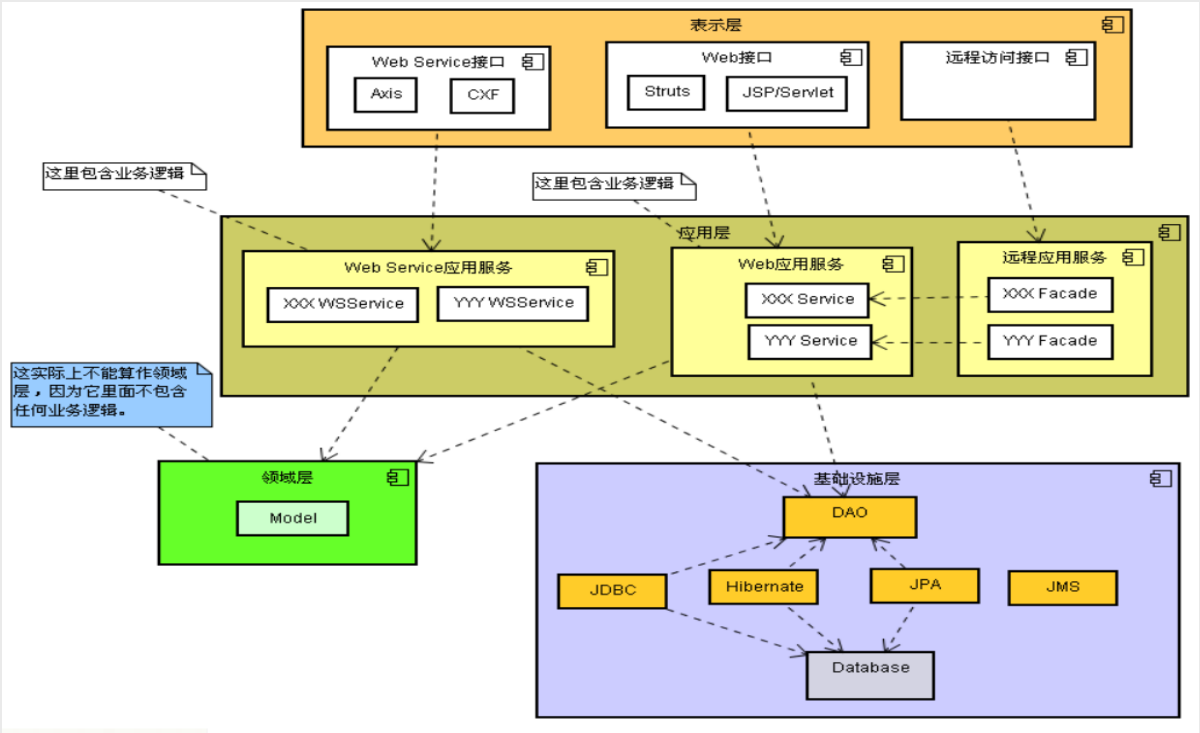
补充：防火墙的架设是 Web 安全的重要保障，ModSecurity 是开源的 Web 防火墙中的佼佼者。企业级防火墙的架设应当有两级防火墙，Web 服务器和部分应用服务器可以架设在两级防火墙之间的 DMZ，而数据和资源服务器应当架设在第二级防火墙之后。

169. 什么是领域模型(domain model)？贫血模型(anaemic domain model)和充血模型(rich domain model)有什么区别？

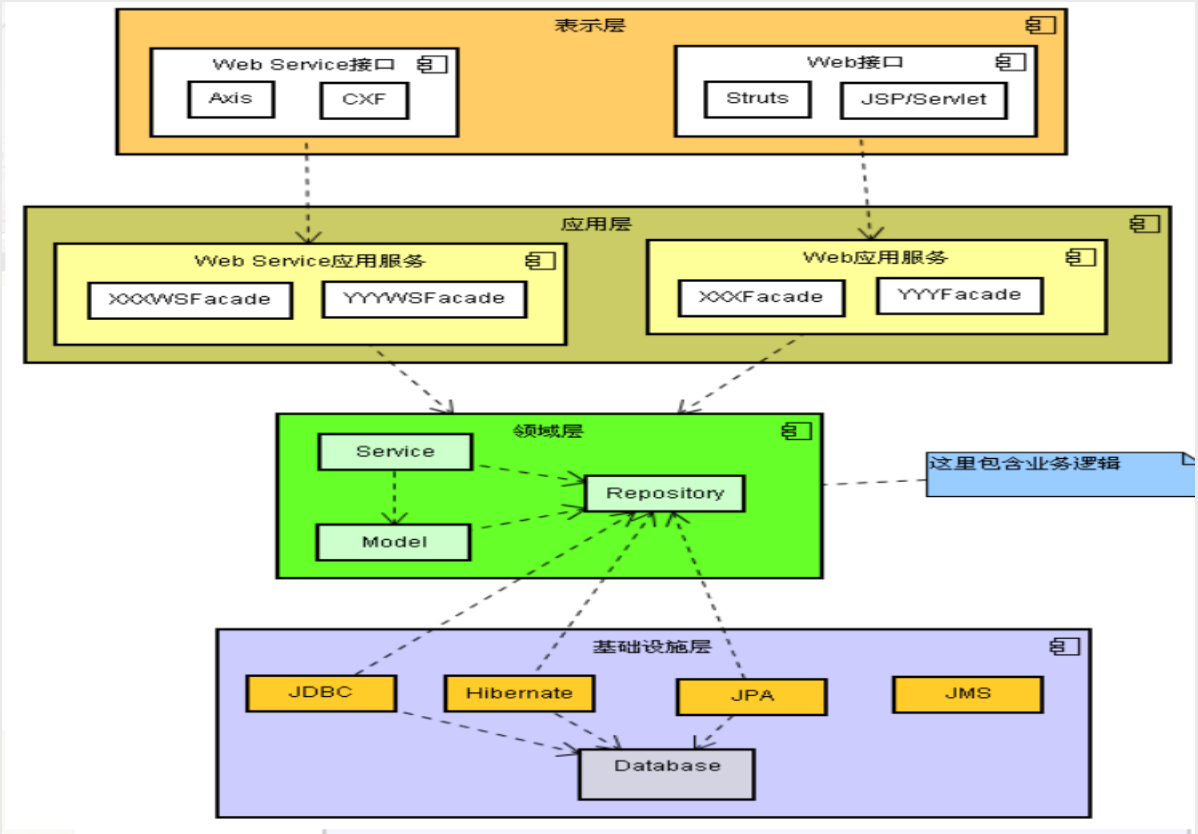
答：领域模型是领域内的概念类或现实世界中对象的可视化表示，又称为概念模型或分析对象模型，它专注于分析问题领域本身，发掘重要的业务领域概念，并建立业务领域概念之间的关系。贫血模型是指使用的领域对象中只有 setter 和 getter 方法（POJO），所有的业务逻辑都不包含在领域对象中而是放在业务逻辑层。有人将我们这里说的贫血模型进一步划分成失血模型（领域对象完全没有业务逻辑）和贫血模型（领域对象有少量的业务逻辑），我们这里就不对此加以区分了。充血模型将大多数业务逻辑和持久化放在领域对象

中，业务逻辑（业务门面）只是完成对业务逻辑的封装、事务和权限等的处理。下面两张图分别展示了贫血模型和充血模型的分层架构。

贫血模型



充血模型



贫血模型下组织领域逻辑通常使用事务脚本模式，让每个过程对应用户可能要做的一个动作，每个动作由一个过程来驱动。也就是说在设计业务逻辑接口的时候，每个方法对应着用户的一个操作，这种模式有以下几个有点：
- 它是一个大多数开发者都能够理解的简单过程模型（适合国内的绝大多数开发者）。
- 它能够与一个使用行数据入口或表数据入口的简单数据访问层很好的协作。
- 事务边界的显而易见，一个事务开始于脚本的开始，终止于脚本的结束，很容易通过代理（或切面）实现声明式事务。
然而，事务脚本模式的缺点也是很多的，随着领域逻辑复杂性的增加，系统的复杂性将迅速增加，程序结构将变得极度混乱。开源中国社区上有一篇很好的译文《贫血领域模型是如何导致糟糕的软件产生》对这个问题做了比较细致的阐述。

170. 谈一谈测试驱动开发（TDD）的好处以及你的理解。
答：TDD 是指在编写真正的功能实现代码之前先写测试代码，然后根据需要重构实现代码。在 JUnit 的作者 Kent Beck 的大作《测试驱动开发：实战与模式解析》（Test-Driven Development: by Example）一书中有这么一段内容：“消除恐惧和不确定性是编写测试驱动代码的重要原因”。因为编写代码时的恐惧会让你小心试探，让你回避沟通，让你羞于得到反馈，让你变得焦躁不安，而 TDD 是消除恐惧、让 Java 开发者更加自信更加乐于沟通的重要手段。TDD 会带来的好处可能不会马上呈现，但是你在某个时候一定会发现，这些好处包括：
- 更清晰的代码 — 只写需要的代码
- 更好的设计
- 更出色的灵活性 — 鼓励程序员面向接口编程
- 更快速的反馈 — 不会到系统上线时才知道 bug 的存在

补充：敏捷软件开发的概念已经有很多年了，而且也部分的改变了软件开发这个行业，TDD 也是敏捷开发所倡导的。

TDD 可以在多个层级上应用，包括单元测试（测试一个类中的代码）、集成测试（测试类之间的交互）、系统测试（测试运行的系统）和系统集成测试（测试运行的系统包括使用的第三方组件）。TDD 的实施步骤是：红（失败测试）- 绿（通过测试） - 重构。
在使用 TDD 开发时，经常会遇到需要被测对象需要依赖其他子系统的情况，但是你希望将测试代码跟依赖项隔离，以保证测试代码仅仅针对当前被测对象或方法展开，这时候你需要的是测试替身。测试替身可以分为四类：
- 虚设替身：只传递但是不会使用到的对象，一般用于填充方法的参数列表
- 存根替身：总是返回相同的预设响应，其中可能包括一些虚设状态
- 伪装替身：可以取代真实版本的可用版本（比真实版本还是会差很多）
- 模拟替身：可以表示一系列期望值的对象，并且可以提供预设响应
Java 世界中实现模拟替身的第三方工具非常多，包括 EasyMock、Mockito、jMock 等。