第2章 Java核心基础面试题汇总

2.1 Java中提供了抽象类还有接口，开发中如何去选择呢？

这道题想考察什么？

Java是面向对象编程的，抽象是它的一大特征，而体现这个特征的就是抽象类与接口。抽象类与接口某些情况下都能够互相替代，但是如果真的都能够互相替代，那Java为何会设计出抽象与接口的概念？这就需要面试者能够掌握两者的区别。

考察的知识点

OOP（面向对象）编程思想，抽象与接口的区别与应用场景；

考生应该如何回答

抽象类的设计目的，是代码复用；接口的设计目的，是对类的行为进行约束。

- 当需要表示is-a的关系，并且需要代码复用时用抽象类
- 当需要表示has-a的关系，可以使用接口

比如狗具有睡觉和吃饭方法，我们可以使用接口定义：

```
public interface Dog {
    public void sleep();
    public void eat();
}
```

但是我们发现如果采用接口，就需要让每个派生类都实现一次sleep方法。此时为了完成对sleep方法的复用，我们可以选择采用抽象类来定义：

```
public abstract class Dog {
    public void sleep(){
        //......
    }
    public abstract void eat();
}
```

但是如果我们训练，让狗拥有一项技能——握手，怎么添加这个行为？将握手方法写入抽象类中，但是这会导致所有的狗都能够握手。

握手是训练出来的，是对狗的一种扩展。所以这时候我们需要单独定义握手这种行为。这种行为是否可以采用抽象类来定义呢？

```
public abstract Handshake{
    abstract void doHandshake();
}
```

如果采用抽象类定义握手，那我们现在需要创建一类能够握手的狗怎么办？

```
public class HandShakeDog extends Dog //,Handshake
```

大家都知道在Java中不能多继承，这是因为多继承存在二义性问题。

> 二义性问题：一个类如果能够继承多个父类，那么如果其中父类A与父类B具有相同的方法，当调用这个方法时会调用哪个父类的方法呢？

所以此时，我们就需要使用接口来定义握手行为：

```
public interface Handshake{
    void doHandshake();
}
public class HandShakeDog extends Dog implements Handshake
```

不是所有的狗都会握手，也不止狗会握手。我们可以同样训练猫，让猫也具备握手的技能，那么猫Cat类，同样能够实现此接口，这就是"has-a"的关系。

抽象类强调从属关系，接口强调功能，除了使用场景的不同之外，在定义中的不同则是：

类型	abstract class	Interface
定义	abstract class关键字	Interface关键字
继承	抽象类可以继承一个类和实现多个接口；子类只可以继承一个抽象类	接口只可以继承接口（一个或多个）；子类可以实现多个接口
访问修饰符	抽象方法可以有**public**、**protected**和**default**这些修饰符	接口方法默认修饰符是**public**。你不可以使用其它修饰符
方法实现	可定义构造方法，可以有抽象方法和具体方法	接口完全是抽象的，没构造方法，且方法都是抽象的，不存在方法的实现
实现方式	子类使用**extends**关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现	子类使用关键字**implements**来实现接口。它需要提供接口中所有声明的方法的实现
作用	了把相同的东西提取出来,**即重用**	为了把程序模块进行固化的契约,是**为了降低偶合**

2.2 重载和重写是什么意思，区别是什么？（京东）

这道题想考察什么？

Java基础

考察的知识点

面向对象多态的基础概念

考生应该如何回答

重写(Override)

重写就是重新写的意思，当父类中的方法对于子类来说不适用或者需要扩展增强时，子类可以对从父类中继承来的方法进行重写。

比如Activity是Android开发中四大组件之一。在Activity中存在各种声明周期方法:onCreate、onStart.....等等。而我们应用中需要使用Activity来展示UI，那么我们会需要编写自己的类继承自Activity。

```
public class MainActivity extends Activity{
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

在上述代码中，onCreate就是被我们重写的Activity中定义方法。我们在onCreate增加了`setContentView`

的调用，完成了对超类Activity中对应方法的修改与扩展。

重载(Overload)

重载则是在同一个类中，允许存在多个同名方法，只要它们的参数列表不同即可。比如在Android开发中，我们会使用LayoutInflater的inflate方法加载布局，inflate方法存在多个定义，其中包括两个参数的，与三个参数的：

```
public View inflate(@LayoutRes int resource, @Nullable ViewGroup root) {
    return inflate(resource, root, root != null);
}
public View inflate(@LayoutRes int resource, @Nullable ViewGroup root, boolean
attachToRoot) {
    //......
}
```

可以看到在两个参数的inflate方法中，会为我们调起三个参数的inflate方法，而定义第一个inflate方法的目的就是为了提供默认的`attachToRoot`参数值。因为Java中无法定义方法参数默认值，所以我们经常使用重载达成此目的。

2.3 静态内部类是什么？和非静态内部类的区别是什么？

这道题想考察什么？

掌握static的作用与注意事项

考察的知识点

Java中关键字static

考生应该如何回答

在定义内部类时，如果内部类被static声明，则该内部类为静态内部类。

```
public class OuterClass{
    static class InnerClass{

    }
}
```

当内部类被static声明，那么在内部类中就无法直接使用外部类的属性。比如编写普通内部类：

```
public class OuterClass{
    int i;
    public class InnerClass{
        public InnerClass(){
            i = 10;
        }
    }
}
```

此时对OuterClass.java 进行编译，会生成：OuterClass.class 与 OuterClass$InnerClass.class 两个文件。对后者反编译我们将看到：

```
public class OuterClass$InnerClass {
    public OuterClass$InnerClass(OuterClass var1) {
        this.this$0 = var1;
        var1.i = 10;
    }
}
```

可以看到，普通内部类构造方法中实际上会隐式的传递外部类实例对象给内部类。在内部类中使用外部类的属性：i。实际上是通过外部类的实例对象：var1获取的。同时如果我们需要构建出 `InnerClass` 的实例对象，非静态内部类也无法脱离外部类实体被创建。

下面我们将InnerClass定义为static静态内部类：

```
public class OuterClass{
    int i;
    public static class InnerClass{
        public InnerClass(){
            //i = 10;
        }
    }
}
```

此时无法使用外部类的普通成员属性：i。其对应字节码为：

```
public class OuterClass$InnerClass {
    public OuterClass$InnerClass() {

    }
}
```

静态内部类中不再隐式的持有外部类的实例对象。但是如果我们将属性i定义为static，那么在静态内部类中也是可以直接使用外部类的静态成员属性的，此时字节码为：

```
public class OuterClass$InnerClass {
    public OuterClass$InnerClass() {
        OuterClass.i = 10;
    }
}
```

内部静态类不需要有指向外部类的引用。但非静态内部类需要持有对外部类的引用。但是静态内部类能够直接利用 `new OuterClass.InnerClass()` 实例化。

因此静态内部类与非静态内部类的区别有：

1. 非静态内部类能够访问外部类的静态和非静态成员，静态类只能访问外部类的静态成员。
2. 非静态内部类不能脱离外部类被创建，静态内部类可以。

2.4 Java中在传参数时是将值进行传递，还是传递引用？

这道题想考察什么？

是否了解什么是值传递和引用传递与真实场景使用，是否熟悉什么是值传递和引用传递在工作中的表现是什么？

考察的知识点

什么是值传递和引用传递的概念，两者对开发中编写的代码的影响

考生应该如何回答

值传递：在方法调用时，传递的参数是这个参数指向值的拷贝；

引用传递：在方法调用时，传递引用的地址

在Java中对于参数的传递可以分为两种情况：

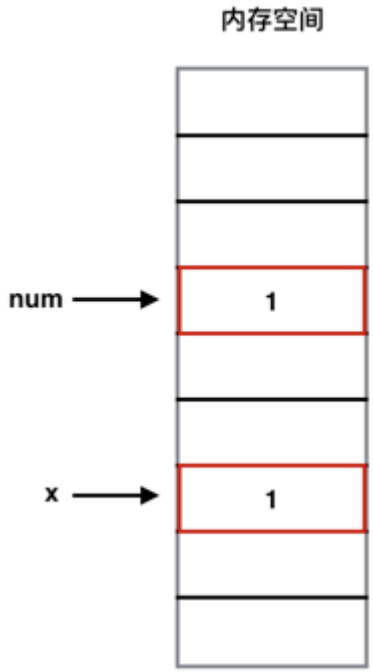
1.基本数据类型的参数

```
1 public class TransferTest {
2     public static void main(String[] args) {
3         int num = 1;
4         System.out.println("changeNum()方法调用之前：num = " + num);
5         changeNum(num);
6         System.out.println("changeNum()方法调用之后：num = " + num);
7     }
8 
9     public static void changeNum(int x) {
10        x = 2;
11    }
12 }
```

运行结果：

```
changeNum()方法调用之前: num = 1
changeNum()方法调用之后: num = 1
```

传递过程的示意图如下：



分析：num作为参数传递给changeNum()方法时，是将内存空间中num所指向的那个存储单元中存放的值1复制了一份传递给了changeNum()方法中的x变量，而这个x变量也在内存空间中分配的一个存储单元。这时就把num对的值1传递给了x变量所指向的存储单元中。此后在changeNum()方法中对x变量的一切操作都是针对于x所指向的这个存储单元，与num所指向的存储单元无关。

所以，在changeNum()方法被调用后，num所指向的存储单元的值还是没有发生变化，这就是所谓的“值传递”。

值传递的精髓是：传递的是存储单元中的内容，而不是存储单元的引用。

2.引用类型的参数

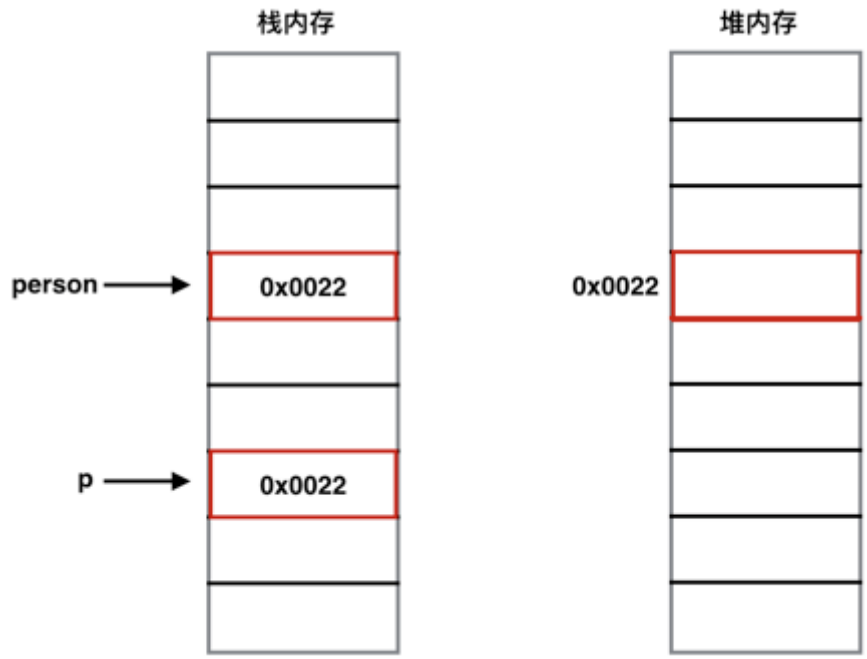
```
1  public class TransferTest2 {
2      public static void main(String[] args) {
3          Person person = new Person();
4          System.out.println(person);
5          change(person);
6          System.out.println(person);
7      }
8
9      public static void change(Person p) {
10         p = new Person();
11     }
12 }
13
14 /**
15  * Person类
16  */
17 class Person {
18
19 }
```

运行结果：

```
Person@1d44bcfa
Person@1d44bcfa
```

可以看出两次打印结果一致。即调用change()方法后，person变量并没发生改变。

传递过程的示意图如下：



分析：

01.当程序执行到第3行 Person person = new Person()时，程序在堆内存（heap）中开辟了一块内存空间用来存储Person类实例对象，同时在栈内存（stack）中开辟了一个存储单元来存储该实例对象的引用，即上图中person指向的存储单元。

02.当程序执行到第5行 change(person)时，person作为参数（实参）传递给了change()方法。这里是person将自己的存储单元的内容传递给了change()方法的p变量。 此后在change()方法中对p变量的一切操作都是针对于p变量所指向的存储单元，与perosn所指向的存储单元就没有关系了。

因此Java的参数传递，不管是基本数据类型还是引用类型的参数，都是按值传递！

2.5 使用equals和==进行比较的区别

这道题想考察什么？

在开发中当需要对引用类型和基本数据类型比较时应该怎么做，为什么有区别。

考察的知识点

equals 的实现以及栈和堆的内存管理。

考生应该如何回答

==

对于不同的数据类型，使用==进行比较的意义不同：

- 基本数据类型（也称原始数据类型）：byte,short,char,int,long,float,double,boolean。用==，比较的是他们的值；
- 引用数据类型：用（==）进行比较的时候，比较的是地址。

对于引用类型，除非是同一个new出来的对象，他们的比较的结果为true，否则为false。因为每new一次，都会重新开辟堆内存空间，哪怕他们的值一致，但是也是在不同的地址存放。所以对于引用类型的值比较应该使用equals方法。

equals()方法介绍

equals 方法是 Object 中定义的一个方法，源码如下：

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

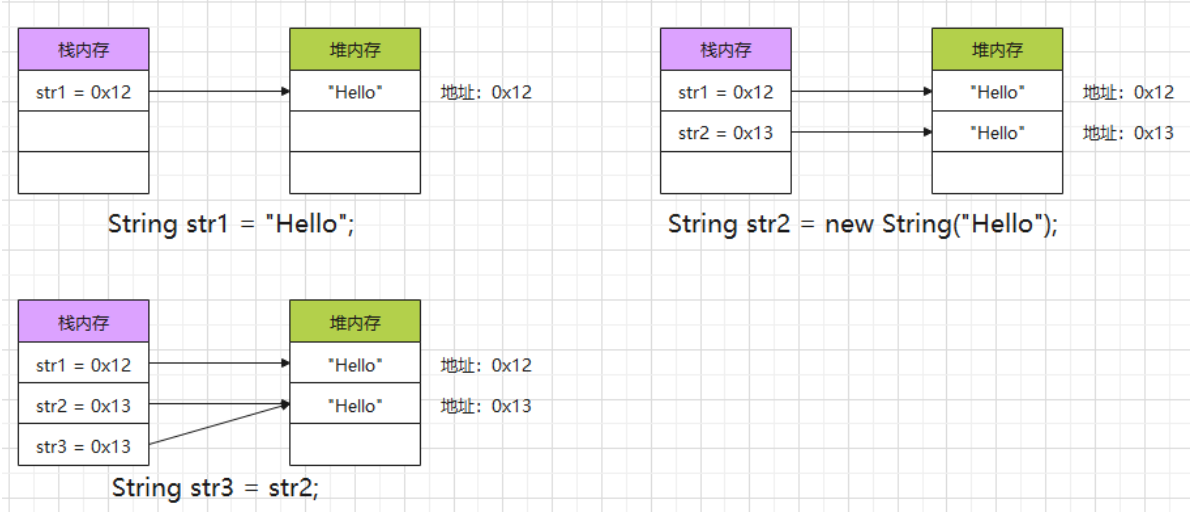
可以看到实际上就是用的 == 实现的，所以这个原始方法意义不大，一般在类中做比较的时候，都会重写这个方法，如String、Integer、Date等。

```
//Integer
public boolean equals(Object obj) {
    if (obj instanceof Integer) {
        //比较Integer中包装的int值
        return value == ((Integer)obj).intValue();
    }
    return false;
}
//String
public boolean equals(Object anObject) {
    if (this == anObject) { //同一个对象
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = length();
        if (n == anotherString.length()) {
            int i = 0;
            while (n-- != 0) {
                if (charAt(i) != anotherString.charAt(i))  //逐个字符比较
                    return false;
                i++;
            }
            return true;
        }
    }
    return false;
}
```

String的比较

```
public class StringDemo {
    public static void main(String args[]) {
        String str1 = "Hello";
        String str2 = new String("Hello");
        String str3 = str2; // 引用传递
        System.out.println(str1 == str2); // false
        System.out.println(str1 == str3); // false
        System.out.println(str2 == str3); // true
        System.out.println(str1.equals(str2)); // true
        System.out.println(str1.equals(str3)); // true
        System.out.println(str2.equals(str3)); // true
    }
}
```

栈和堆的内存分析图：



由此可见，equals 是比较字符串的内容是否一样，== 是比较字符串的堆内存地址是否一样。

结论

equals和==的区别，需要分情况讨论：

1. 没有重写 equals ，则 equals 和 == 是一样的。
2. 如果重写了 equals，则需看 equals 的方法实现。以 String 类为例：
    1. equals 是比较字符串的内容是否一样；
    2. == 是比较字符串的堆内存地址是否一样，或者说引用的值是否相等。

2.6 String s = new String("xxx");创建了几个String对象?

这道题想考察什么？

在开发中常用的字符串String

考察的知识点

Java基础，JVM常量池与对象内存分配

考生应该如何回答

首先代码`String s = new String("xxx")`中包含关键字`new`，我们都知道此关键字是创建类的实例对象。JVM在运行期执行new指令因此这会在堆中创建一个String对象。

其次，在String的构造方法中传递了"xxx"字符串，此处的"xxx"是一个字符串常量。JVM会首先从字符串常量池中尝试获取其对应的引用。如果不存在，则会在堆中创建"xxx"的字符串对象，并将其引用保存到字符串常量池然后返回。

因此，如果`xxx`这个字符串常量不存在，则创建两个String对象；而如果存在则只会创建一个String对象。

2.7 finally中的代码一定会执行吗？try里有return，finally还执行么

这道题想考察什么？

对Java语言的深层次理解，避免在开发时写出"问题"代码

考察的知识点

JVM执行流程

考生应该如何回答

在Java官方文档中对finally的描述如下：

```
The `finally` block *always* executes when the `try` block exits.
```

大致意思是：finally代码块中的内容一定会得到执行。

JVM规范里面同样也有明确说明

```
If the try clause executes a return, the compiled code does the following:

1. Saves the return value (if any) in a local variable.
2. Executes a jsr to the code for the finally clause.
3. Upon return from the finally clause, returns the value saved in the local
variable.
```

意思是如果在try中存在return的情况下，会把try中return的值存到栈帧的局部变量表中，然后去执行finally语句块，最后再从局部变量表中取回return的值返回。另外，当try和finally里都有return时，会忽略try的return，而使用finally的return。

特殊情况

在正常情况下，finally中的代码一定会得到执行，但是如果我们将执行try-catch-finally 代码块的线程设置为守护线程，或者在fianlly之前调用`System.exit`结束当前虚拟机，那么finally则不会得到执行：

```
try{
    System.exit(0);
}catch (Exception e){

}finally {

}

Thread t1 = new Thread(){
    @Override
    public void run(){
        //try-catch-finally
    }
};
t1.setDaemon(true);//设置为守护进程
t1.start();
```

2.8 Java异常机制中，异常Exception与错误Error区别

这道题想考察什么？

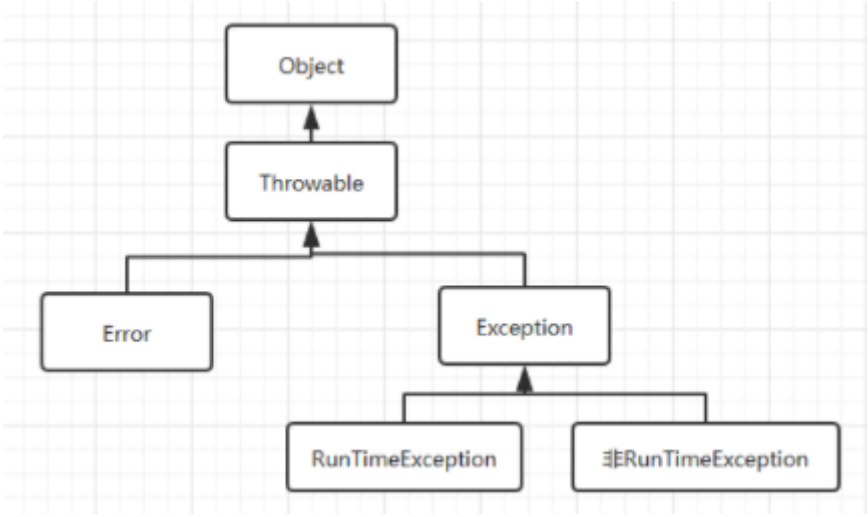
在开发时需要时候需要自定义异常时，应该选择定义Excption还是Error？编写的代码触发Excption或者Error分别代表什么？

考察的知识点

Java异常机制

考生应该如何回答

在Java中存在一个`Throwable`可抛出类，`Throwable`有两个重要的子类，一个是Error，另一个则是Exception。



`Error`是程序不能处理的错误，表示程序中较严重问题。例如，Java虚拟机运行错误（Virtual MachineError），当 JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError等等。这些错误发生时，JVM一般会选择线程终止。这些错误是不可查的，它们在程序的控制和处理能力之外，而且绝大多数是程序运行时不允许出现的状况。

`Exception`是程序可以处理的异常。而Exception又分为运行时异常（RuntimeException）与非运行时异常。

- 运行异常

  运行时异常，**又称不受检查异常**。所谓的不受检查，指的是Java编译检查时不会告诉我们有这个异常，需要在运行时候才会暴露出来，比如下标越界，空指针异常等。

- 非运行时异常

  RuntimeException之外的异常我们统称为**非运行时异常**，比如IOException、SQLException，是必须进行处理的异常（**检查异常**），如果不处理（throw到上层或者try-catch），程序就不能编译通过。

2.9 序列Parcelable,Serializable的区别？(阿里)

详细讲解

享学课堂移动互联网系统课程：架构师筑基必备技能《NDK专题-JNI实战篇》

这道题想考察什么？

掌握序列化接口实现原理，针对不同场景在工作中合理运用

考察的知识点

Parcelable原理

Serializable原理

考生应该如何回答

序列化 是将对象的状态信息转换为可以存储或传输的形式的过程。在序列化期间，对象将其当前状态写入到临时或持久性存储区。以后，可以通过从存储区中读取或反序列化对象的状态，重新创建该对象。Serializable是Java提供的序列化机制，而 Parcelable则是Android提供的适合于内存中进行传输的序列化方式。

Serializable

Serializable是Java中提供的一个 序列化接口，然而这个接口并没有任何方法需要实现者实现。

```
public interface Serializable {
}
```

这表示Serializable接口知识用来标识当前类可以被序列化与反序列化。

基本使用

实现了Serializable接口的类对象能够通过 ObjectOutputStream 将需要序列化数据写入到流中，因为 Java IO 是一种装饰者模式，因此可以通过 ObjectOutStream 包装 FileOutStream 将数据写入到文件中或者包装 ByteArrayOutStream 将数据写入到内存中。也可以通过 ObjectInputStream 从磁盘或者内存读取数据然后转化为指定的对象即可(反序列化)。

```
try {
    TestBean serialization = new TestBean("a","b");
    //序列化
    ObjectOutputStream os = new ObjectOutputStream(new
FileOutputStream("/path/xxx"));
    os.writeObject(serialization);
    //反序列化
    ObjectInputStream is = new ObjectInputStream(new
FileInputStream("/path/xxx"));
    TestBean deserialization = (TestBean) is.readObject();
} catch (IOException e) {
    e.printStackTrace();
}
```

实现了Serializable接口的类中成员属性除基本数据类型外，即引用类型，也需要实现Serializable接口，否则将在序列化时抛出`java.io.NotSerializableException`异常。

```
public class TestBean implements Serializable {

    private String name;
    private String pwd;
    //Gson未实现Serializable接口，TestBean实例对象在序列化时抛出
NotSerializableException
    private Gson gson = new Gson();
    public TestBean(String name, String pwd) {
        this.name = name;
        this.pwd = pwd;
    }

    public TestBean() {
    }

    public String getName() {
        return name;
    }

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

```
    public String getPwd() {
        return pwd;
    }

    public void setPwd(String pwd) {
        this.pwd = pwd;
    }

}
```

因此，如果需要对某个类进行序列化时，对于类中不需要进行序列化与反序列化的属性，可以使用 **transient** 关键字声明。

```
private transient Gson gson = new Gson();
```

serialVersionUID

在实现了Serializable接口的类中，应该为此类定义一个serialVersionUID属性：

```
private static final long serialVersionUID = 1L;
```

虽然不定义程序依然能够正常运行。 但是Java序列化的机制是通过判断类的serialVersionUID来验证的版本一致的。

序列化操作时会把当前类的serialVersionUID写入到序列化文件中，当反序列化时系统会自动检测文件中的serialVersionUID，判断它是否与当前类中的serialVersionUID一致。如果一致说明序列化文件的版本与当前类的版本是一样的，可以反序列化成功，否则就失败。

例如在A程序中TestBean类的serialVersionUID=1L，而在程序B中TestBean类的serialVersionUID=2L。 那么A程序序列化输出的数据，在B程序中就无法反序列化为TestBean对象。

Parcelable

Parcelable是Android为我们提供的序列化的接口。相对于Java的Serializable，Parcelable的使用稍显复杂：

```
public class TestBean implements Parcelable {

    private String name;
    private String pwd;

    public TestBean(String name, String pwd) {
        this.name = name;
        this.pwd = pwd;
    }

    public TestBean() {
    }

    public int describeContents() {
        return 0;
    }
    //序列化
    public void writeToParcel(Parcel out, int flags) {
```

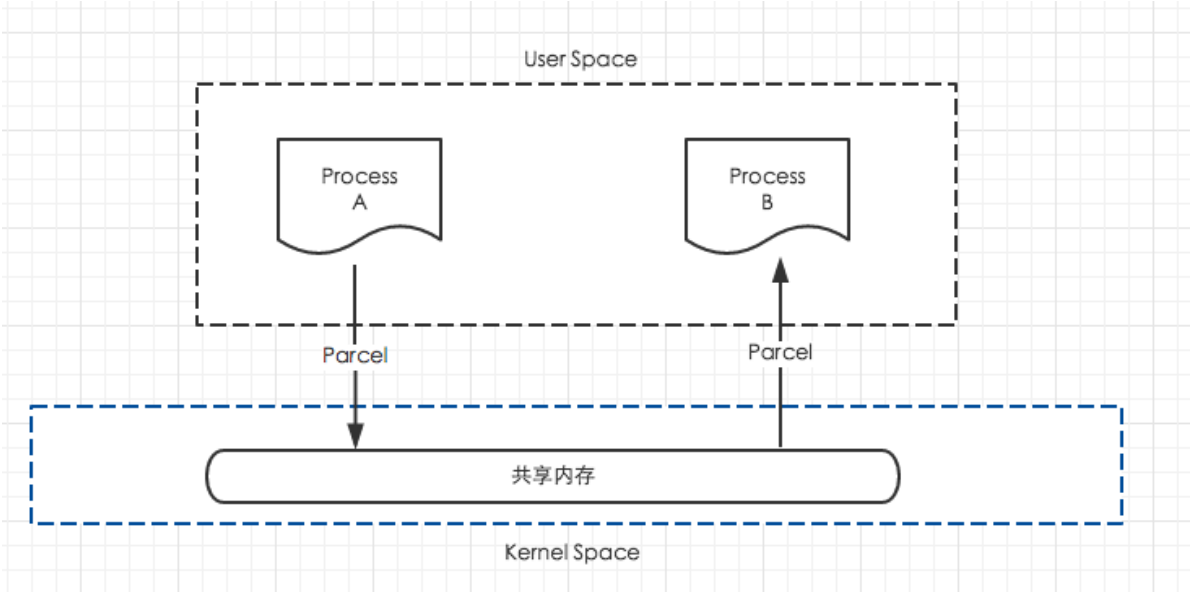
```
        out.writeString(name);
         out.writeString(pwd);
    }
    public static final Parcelable.Creator<TestBean> CREATOR
        = new Parcelable.Creator<TestBean>(){
            public TestBean createFromParcel(Parcel in) {
                return new TestBean(in);
            }

            public MyParcelable[] newArray(int size) {
                return new TestBean[size];
            }
    }
    private TestBean(Parcel in) {
        // 反序列化：按序列化顺序读取
        name = in.readString();
        pwd = in.readString();
    }
}
```

Parcelable接口的实现类是通过Parcel写入和恢复数据的,并且必须要有一个非空的静态变量 CREATOR。

Parcel

Parcel其实就是一个数据载体，可以将序列化之后的数据写入到一个共享内存中，其他进程通过Parcel可以从这块共享内存中读出字节流，并反序列化成对象。



Parcel可以读写原始数据类型，也可以读写实现了Parcelable对象。

因此Parcelable实现序列化的原理就是将一个完整的对象进行分解，而分解后的每一部分都是基本数据类型或者其他实现了Parcelable/Serializable的类型，从而实现传递对象的功能。

区别

Parcelable和Serializable都是实现序列化并且在Android中都可以使用Intent进行数据传递。Serializable是Java的实现方式，实现过程中会频繁的IO操作，但是实现方式简单。而Parcelable是Android提供的方式，效率比较高（号称比Serializable快10倍），但是实现起来复杂一些 。

如果只需要在内存中进行数据传输是，序列化应该选择Parcelable，而如果需要存储到设备或者网络传输上应该选择Serializable。这是因为Android不同版本Parcelable数据规则可能不同，所以不推荐使用Parcelable进行数据持久化。

2.10 为什么Intent传递对象为什么需要序列化？(阿里)

这道题想考察什么？

掌握序列化的意义与Android数据传输的原理

考察的知识点

序列化

Binder

考生应该如何回答

在Android中使用Intent传输数据除了基本数据类型之外，对于其他类型对象需要此类型实现了Serializable或者Parcelable序列化接口才能进行传输。

以startActivity为例：

```
Intent intent = new Intent(context,OtherActivity.class);
//字符串实现了Serializable序列化
intent.putExtra("a","享学");
//Message实现了Parcelable序列化
intent.putExtra("b",new android.os.Message());
//错误：上下文context并未实现序列化
intent.putExtra("c",context);
startActivity(intent);
```

Intent传输数据本质上是使用Binder来完成的。Intent启动组件需要借助AMS完成，因此`startActivity`会离开当前应用进程，进入AMS所在的system_server进程进行跨进程通信。这就意味着传输的对象需要在不同进程之间进行传输。

为了保护不同进程互不干扰，进程隔离让system_server进程无法直接获取应用进程内存中的对象。因此必须通过类似于复制的手段，将应用进程的对象传递给system_server进程，再由system_server进程传递给应用中的`OtherActivity`。

根据《**2.9 序列Parcelable,Serializable的区别？**》可知，Serializable会利用IO将对象写入到文件中；而Parcelable则会将对象写入到Parcel中，两种方式都可以解决跨进程的数据传递。因此Intent传递的对象需要实现Serializable或者Parcelable序列化。