序列化

定义以及相关概念

1. 由于在系统底层，数据的传输形式是简单的字节序列形式传递，即在底层，系统不认识对象，只认识字节序列，而为了达到进程通讯的目的，需要先将数据序列化，而序列化就是将对象转化字节序列的过程。相反地，当字节序列被运到相应的进程的时候，进程为了识别这些数据，就要将其反序列化，即把字节序列转化为对象
2. 无论是在进程间通信、本地数据存储又或者是网络数据传输都离不开序列化的支持。而针对不同场景选择合适的序列化方案对于应用的性能有着极大的影响。
3. 从广义上讲，数据序列化就是将数据结构或者是对象转换成我们可以存储或者传输的数据格式的一个过程，在序列化的过程中，数据结构或者对象将其状态信息写入到临时或者持久性的存储区中，而在对应的反序列化过程中，则可以说是生成的数据被还原成数据结构或对象的过程。
4. 这样来说，数据序列化相当于是将我们原先的对象序列化概念做出了扩展，在对象序列化和反序列化中，我们熟知的有两种方法，其一是Java语言中提供的Serializable接口，其二是Android提供的Parcelable接口。而在这里，因为我们对这个概念做出了扩展，因此也需要考虑几种专门针对数据结构进行序列化的方法，如现在那些个开放API一般返回的数据都是JSON格式的，又或者是我们Android原生的SQLite数据库来实现数据的本地存储，从广义上来说，这些都可以算做是数据的序列化

序列化

将数据结构或对象转换成二进制串的过程。

反序列化

将在序列化过程中所生成的二进制串转换成数据结构或者对象的过程

数据结构、对象与二进制串

不同的计算机语言中，数据结构，对象以及二进制串的表示方式并不相同。

数据结构和对象：对于类似 Java 这种完全面向对象的语言，工程师所操作的一切都是对象（Object），来自于类的实例化。在 Java 语言中最接近数据结构的概念，就是 POJO（Plain Old Java Object）或者 Javabean - - 那些只有 setter/getter 方法的类。而在 C 二进制串：序列化所生成的二进制串指的是存储在内存中的一块数据。C 语言的字符串可以直接被传输层使用，因为其本质上就是以'0'结尾的存储在内存中的二进制串。在 Java 语言里面，二进制串的概念容易和 String 混淆。实际上 String 是 Java 的一等公民，是一种特殊对象（Object）。对于跨语言间的通讯，序列化后的数据当然不能是某种语言的特殊数据类型。二进制串在 Java 里面所指的是 byte[]，byte 是 Java 的 8 中原生数据类型之一（Primitive data types）。

序列化/反序列化的目的

简单的概括

- 序列化：主要用于网络传输，数据持久化，一般序列化也称为编码(Encode)
- 反序列化：主要用于从网络，磁盘上读取字节数组还原成原始对象，一般反序列化也称为解码(Decode)

具体的讲：

- 永久的保存对象数据(将对象数据保存在文件当中,或者是磁盘中)

- 通过序列化操作将对象数据在网络上进行传输(由于网络传输是以字节流的方式对数据进行传输的.因此序列化的目的是将对象数据转换成字节流的形式)
- 将对象数据在进程之间进行传递(Activity之间传递对象数据时,需要在当前的Activity中对对象数据进行序列化操作.在另一个Activity中需要进行反序列化操作讲数据取出)
- Java平台允许我们在内存中创建可复用的Java对象，但一般情况下，只有当JVM处于运行时，这些对象才可能存在，即，这些对象的生命周期不会比JVM的生命周期更长（即每个对象都在JVM中）但在现实应用中，就可能要停止JVM运行，但有要保存某些指定的对象，并在将来重新读取被保存的对象。这是Java对象序列化就能够实现该功能。（可选择入数据库、或文件的形式保存）
- 序列化对象的时候只是针对变量进行序列化,不针对方法进行序列化.
- 在Intent之间,基本的数据类型直接进行相关传递即可,但是一旦数据类型比较复杂的时候,就需要进行序列化操作了.

序列化协议特性

通用性

- 技术层面，序列化协议是否支持跨平台、跨语言。如果不支持，在技术层面上的通用性就大大降低了。
- 流行程度，序列化和反序列化需要多方参与，很少人使用的协议往往意味着昂贵的学习成本；另一方面，流行度低的协议，往往缺乏稳定而成熟的跨语言、跨平台的公共包。

强健性 / 鲁棒性

- 成熟度不够
- 语言 / 平台的不公平性

可调试性 / 可读性

- 支持不到位
- 访问限制

性能

性能包括两个方面，时间复杂度和空间复杂度。

- 空间开销（Verbosity）， 序列化需要在原有的数据上加上描述字段，以为反序列化解析之用。如果序列化过程引入的额外开销过高，可能会导致过大的网络，磁盘等各方面的压力。对于海量分布式存储系统，数据量往往以 TB 为单位，巨大的的额外空间开销意味着高昂的成本。
- 时间开销（Complexity），复杂的序列化协议会导致较长的解析时间，这可能会使得序列化和反序列化阶段成为整个系统的瓶颈。

可扩展性 / 兼容性

移动互联时代，业务系统需求的更新周期变得更快，新的需求不断涌现，而老的系统还是需要继续维护。如果序列化协议具有良好的可扩展性，支持自动增加新的业务字段，而不影响老的服务，这将大大提供系统的灵活度。

安全性 / 访问限制

在序列化选型的过程中，安全性的考虑往往发生在跨局域网访问的场景。当通讯发生在公司之间或者跨机房的时候，出于安全的考虑，对于跨局域网的访问往往被限制为基于 HTTP/HTTPS 的 80 和 443 端口。如果使用的序列化协议没有兼容而成熟的 HTTP 传输层框架支持，可能会导致以下三种结果之一：

- 因为访问限制而降低服务可用性；

- 被迫重新实现安全协议而导致实施成本大大提高；
- 开放更多的防火墙端口和协议访问，而牺牲安全性
- **注意点：Android的Parcelable也有安全漏洞**

> 最近几个月，Android安全公告公布了一系列系统框架层的高危提权漏洞，如下表所示。

参考

> https://www.anquanke.com/post/id/103570

几种常见的序列化和反序列化协议

XML&SOAP

XML 是一种常用的序列化和反序列化协议，具有跨机器，跨语言等优点，SOAP（Simple Object Access protocol） 是一种被广泛应用的，基于 XML 为序列化和反序列化协议的结构化消息传递协议

JSON（Javascript Object Notation）

JSON 起源于弱类型语言 Javascript， 它的产生来自于一种称之为"Associative array"的概念，其本质是就是采用"Attribute - value"的方式来描述对象。实际上在 Javascript 和 PHP 等弱类型语言中，类的描述方式就是 Associative array。JSON 的如下优点，使得它快速成为最广泛使用的序列化协议之一。

- 这种 Associative array 格式非常符合工程师对对象的理解。
- 它保持了 XML 的人眼可读（Human-readable）的优点。
- 相对于 XML 而言，序列化后的数据更加简洁。 来自于的以下链接的研究表明：XML 所产生序列化之后文件的大小接近 JSON 的两倍
- 它具备 Javascript 的先天性支持，所以被广泛应用于 Web browser 的应用常景中，是 Ajax 的事实标准协议。
- 与 XML 相比，其协议比较简单，解析速度比较快。
- 松散的 Associative array 使得其具有良好的可扩展性和兼容性

Protobuf

Protobuf 具备了优秀的序列化协议的所需的众多典型特征。

- 标准的 IDL 和 IDL 编译器，这使得其对工程师非常友好。
- 序列化数据非常简洁，紧凑，与 XML 相比，其序列化之后的数据量约为 1/3 到 1/10。
- 解析速度非常快，比对应的 XML 快约 20-100 倍。
- 提供了非常友好的动态库，使用非常简介，反序列化只需要一行代码。

Android程序员该如何选择序列化方案

Serializable接口

是 Java 提供的序列化接口，它是一个空接口：

```
1 public interface Serializable {
2 }
```

Serializable 用来标识当前类可以被 ObjectOutputStream 序列化，以及被 ObjectInputStream 反序列化。

Serializable入门

```
public class Student implements Serializable {
    //serialVersionUID唯一标识了一个可序列化的类
    private static final long serialVersionUID = -2100492893943893602L;
    private String name;
    private String sax;
    private Integer age;

    //Course也需要实现Serializable接口
    private List<Course> courses;

    //用transient关键字标记的成员变量不参与序列化(在被反序列化后，transient 变量的值被设为初始值，如 int 型的是 0，对象型的是 null)
    private transient Date createTime;
    //静态成员变量属于类不属于对象，所以不会参与序列化(对象序列化保存的是对象的“状态”，也就是它的成员变量，因此序列化不会关注静态变量)
    private static SimpleDateFormat simpleDateFormat = new SimpleDateFormat();

    public Student() {
        System.out.println("Student: empty");
    }

    public Student(String name, String sax, Integer age) {
        System.out.println("Student: " + name + " " + sax + " " + age);
        this.name = name;
        this.sax = sax;
        this.age = age;
        courses = new ArrayList<>();
        createTime = new Date();
    }
    ...
}

////Course也需要实现Serializable接口
public class Course implements Serializable {

    private static final long serialVersionUID = 667279791530738499L;
    private String name;
    private float score;

    ...
}
```

Serializable 有以下几个特点：

- 可序列化类中，未实现 Serializable 的属性状态无法被序列化/反序列化
- 也就是说，反序列化一个类的过程中，它的非可序列化的属性将会调用无参构造函数重新创建
- 因此这个属性的无参构造函数必须可以访问，否者运行时会报错
- 一个实现序列化的类，它的子类也是可序列化的

serialVersionUID与兼容性

- serialVersionUID的作用
  serialVersionUID 用来表明类的不同版本间的兼容性。如果你修改了此类, 要修改此值。否则以前

用老版本的类序列化的类恢复时会报错: InvalidClassException

- 设置方式
  在JDK中，可以利用JDK的bin目录下的serialver.exe工具产生这个serialVersionUID，对于Test.class，执行命令：serialver Test
- 兼容性问题
  为了在反序列化时，确保类版本的兼容性，最好在每个要序列化的类中加入 private static final long serialVersionUID这个属性，具体数值自己定义。这样，即使某个类在与之对应的对象 已经序列化出去后做了修改，该对象依然可以被正确反序列化。否则，如果不显式定义该属性，这个属性值将由JVM根据类的相关信息计算，而修改后的类的计算 结果与修改前的类的计算结果往往不同，从而造成对象的反序列化因为类版本不兼容而失败。
  不显式定义这个属性值的另一个坏处是，不利于程序在不同的JVM之间的移植。因为不同的编译器实现该属性值的计算策略可能不同，从而造成虽然类没有改变，但是因为JVM不同，出现因类版本不兼容而无法正确反序列化的现象出现

因此 JVM 规范强烈 建议我们手动声明一个版本号，这个数字可以是随机的，只要固定不变就可以。同时最好是 private 和 final 的，尽量保证不变。

Externalizable接口

```
public interface Externalizable extends Serializable {
    void writeExternal(ObjectOutput var1) throws IOException;

    void readExternal(ObjectInput var1) throws IOException,
ClassNotFoundException;
}
```

简单使用

```
public class Course1 implements Externalizable {

    private static final long serialVersionUID = 667279791530738499L;
    private String name;
    private float score;

    ...

    @Override
    public void writeExternal(ObjectOutput objectOutput) throws IOException
{
        System.out.println("writeExternal");
        objectOutput.writeObject(name);
        objectOutput.writeFloat(score);
    }

    @Override
    public void readExternal(ObjectInput objectInput) throws IOException,
ClassNotFoundException {
        System.out.println("readExternal");
        name = (String)objectInput.readObject();
        score = objectInput.readFloat();
    }
...

    public static void main(String... args) throws Exception {
        //TODO:
```

```
        //TODO:
        Course1 course = new Course1("英语", 12f);
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(out);
        oos.writeObject(course);
        course.setScore(78f);
);
        oos.close();

        ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(bs));
        Course1 course1 = (Course1) ois.readObject();
        System.out.println("course1: " + course1);

    }
```

序列化与反序列化 Serializable

Serializable 的序列化与反序列化分别通过 ObjectOutputStream 和 ObjectInputStream 进行

```
/**
     * 序列化对象
     *
     * @param obj
     * @param path
     * @return
     */
    synchronized public static boolean saveObject(Object obj, String path) {
        if (obj == null) {
            return false;
        }
        ObjectOutputStream oos = null;
        try {
            // 创建序列化流对象
            oos = new ObjectOutputStream(new FileOutputStream(path));
           //序列化
            oos.writeObject(obj);
            oos.close();
            return true;
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (oos != null) {
                try {
                    // 释放资源
                    oos.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        return false;
    }

    /**
     * 反序列化对象
     *
```

```
     * @param path
     * @param <T>
     * @return
     */
    @SuppressWarnings("unchecked ")
    synchronized public static <T> T readObject(String path) {
        ObjectInputStream ojs = null;
        try {
            // 创建反序列化对象
            ojs = new ObjectInputStream(new FileInputStream(path));
            // 还原对象
            return (T) ojs.readObject();
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        } finally {
            if(ojs!=null){
                try {
                    // 释放资源
                    ojs.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        return null;
    }
```

Java的序列化步骤与数据结构分析

序列化算法一般会按步骤做如下事情：

- 将对象实例相关的类元数据输出。
- 递归地输出类的超类描述直到不再有超类。
- 类元数据完了以后，开始从最顶层的超类开始输出对象实例的实际数据值。
- 从上至下递归输出实例的数据

格式化后以二进制打开

```
aced 0005 7372 002e 636f 6d2e 7a65 726f
2e73 6572 6961 6c69 7a61 626c 6564 656d
6f2e 7365 7269 616c 697a 6162 6c65 2e53
7475 6465 6e74 e2d9 8cd7 833d f19e 0200
044c 0003 6167 6574 0013 4c6a 6176 612f
6c61 6e67 2f49 6e74 6567 6572 3b4c 0007
636f 7572 7365 7374 0010 4c6a 6176 612f
7574 696c 2f4c 6973 743b 4c00 046e 616d
6574 0012 4c6a 6176 612f 6c61 6e67 2f53
7472 696e 673b 4c00 0373 6178 7100 7e00
0378 7073 7200 116a 6176 612e 6c61 6e67
2e49 6e74 6567 6572 12e2 a0a4 f781 8738
0200 0149 0005 7661 6c75 6578 7200 106a
6176 612e 6c61 6e67 2e4e 756d 6265 7286
ac95 1d0b 94e0 8b02 0000 7870 0000 0012
7372 0013 6a61 7661 2e75 7469 6c2e 4172
7261 794c 6973 7478 81d2 1d99 c761 9d03
0001 4900 0473 697a 6578 7000 0000 0277
0400 0000 0273 7200 2d63 6f6d 2e7a 6572
```

```
6f2e 7365 7269 616c 697a 6162 6c65 6465
6d6f 2e73 6572 6961 6c69 7a61 626c 652e
436f 7572 7365 0942 a76f 5bfc 8343 0200
0246 0005 7363 6f72 654c 0004 6e61 6d65
7100 7e00 0378 7042 b466 6674 0006 e8af
ade6 9687 7371 007e 000a 42b2 999a 7400
06e6 95b0 e5ad a678 7400 045a 6572 6f74
0003 e794 b7
```

- AC ED: STREAM_MAGIC. 声明使用了序列化协议.
- 00 05: STREAM_VERSION. 序列化协议版本.
- 0x73: TC_OBJECT. 声明这是一个新的对象.
- 0x72: TC_CLASSDESC. 声明这里开始一个新Class。
- 00 2e: Class名字的长度.

readObject/writeObject原理分析

以`oos.writeObject(obj)`为例分析

1. ObjectOutputStream的构造函数设置enableOverride = false

```
public ObjectOutputStream(OutputStream out) throws IOException {
       verifySubclass();
       bout = new BlockDataOutputStream(out);
       handles = new HandleTable(10, (float) 3.00);
       subs = new ReplaceTable(10, (float) 3.00);
       enableOverride = false;//enableOverride = false
       ...
   }
```

2. 所以writeObject方法执行的是writeObject0(obj, false);

```
 public final void writeObject(Object obj) throws IOException {
        //enableOverride=false,不走这里
        if (enableOverride) {
            writeObjectOverride(obj);
            return;
        }
        try {//一般情况都走这里
            writeObject0(obj, false);
       ...
    }
```

3. 在writeObject0方法中,代码非常多，看重点

```
/**
     * Underlying writeObject/writeUnshared implementation.
     */
    private void writeObject0(Object obj, boolean unshared)
        throws IOException
       ...

            // remaining cases
            if (obj instanceof String) {
                writeString((String) obj, unshared);
```

```
            } else if (cl.isArray()) {
                writeArray(obj, desc, unshared);
            } else if (obj instanceof Enum) {
                writeEnum((Enum<?>) obj, desc, unshared);
            } else if (obj instanceof Serializable) {
            //看这里
                writeOrdinaryObject(obj, desc, unshared);
            } else {//如果没有实现Serializable接口，会报
NotSerializableException
                if (extendedDebugInfo) {
                    throw new NotSerializableException(
                        cl.getName() + "\n" + debugInfoStack.toString());
                } else {
                    throw new NotSerializableException(cl.getName());
            ...
    }
```

4. 在writeOrdinaryObject(obj, desc, unshared)方法中

```
private void writeOrdinaryObject(Object obj,
                                 ObjectStreamClass desc,
                                 boolean unshared)
    ...
            if (desc.isExternalizable() && !desc.isProxy()) {
            //如果对象实现了Externalizable接口，那么执行
writeExternalData((Externalizable) obj)方法
                writeExternalData((Externalizable) obj);
            } else {//如果对象实现的是Serializable接口，那么执行的是
writeSerialData(obj, desc)
                writeSerialData(obj, desc);
            }
       ...
    }
//这里我们看看writeExternalData

```

5. writeSerialData方法，主要执行方法：defaultWriteFields(obj, slotDesc)

```
/**
     * Writes instance data for each serializable class of given object,
from
     * superclass to subclass.
     * 最终写序列化的方法
     */
    private void writeSerialData(Object obj, ObjectStreamClass desc)
        throws IOException
    {
        ...
            if (slotDesc.hasWriteObjectMethod()) {
                //如果writeObjectMethod != null（目标类中定义了私有的writeObject
方法），那么将调用目标类中的writeObject方法
                ...
                    slotDesc.invokeWriteObject(obj, this);
                 ...
            } else {
```

```
            //如果如果writeObjectMethod == null， 那么将调用默认的defaultWriteFields方法来读取目标类中的属性
                defaultWriteFields(obj, slotDesc);
            }
        }
    }
```

6. 在ObjectStreamClass中,ObjectOutputStream（ObjectInputStream）会寻找目标类中的私有的writeObject（readObject）方法，赋值给变量writeObjectMethod（readObjectMethod）

```
  /**
     * Creates local class descriptor representing given class.
     */
    private ObjectStreamClass(final Class<?> cl) {
        ...

                    if (externalizable) {
                        cons = getExternalizableConstructor(cl);
                    } else {//，在序列化（反序列化）的时候，ObjectOutputStream（ObjectInputStream）
                        // 会寻找目标类中的私有的writeObject（readObject）方法，
                        // 赋值给变量writeObjectMethod（readObjectMethod）
                        cons = getSerializableConstructor(cl);
                        writeObjectMethod = getPrivateMethod(cl, "writeObject",
                            new Class<?>[] { ObjectOutputStream.class },
                            Void.TYPE);
                        readObjectMethod = getPrivateMethod(cl, "readObject",
                            new Class<?>[] { ObjectInputStream.class },
                            Void.TYPE);
                        readObjectNoDataMethod = getPrivateMethod(
                            cl, "readObjectNoData", null, Void.TYPE);
                        hasWriteObjectData = (writeObjectMethod != null);
                    }
                    domains = getProtectionDomains(cons, cl);
                    writeReplaceMethod = getInheritableMethod(
                        cl, "writeReplace", null, Object.class);
                    readResolveMethod = getInheritableMethod(
                        cl, "readResolve", null, Object.class);
                    return null;
                }
            });
        ...
    }

//ObjectStreamClass类中的一个判断方法
boolean hasWriteObjectMethod() {
    requireInitialized();
    return (writeObjectMethod != null);
}
```

Serializable需要注意的坑

- 多引用写入

```
public class Course implements Serializable {

    private static final long serialVersionUID = 667279791530738499L;
    private String name;
    private float score;
...
    public static void main(String... args) throws Exception {
        //TODO:
        //TODO:
        Course course = new Course("英语", 12f);
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(out);
        oos.writeObject(course);
        course.setScore(78f);
//        oos.reset();
        oos.writeUnshared(course);
//        oos.writeObject(course);
        byte[] bs = out.toByteArray();
        oos.close();

        ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(bs));
        Course course1 = (Course) ois.readObject();
        Course course2 = (Course) ois.readObject();
        System.out.println("course1: " + course1);
        System.out.println("course2: " + course2);
    }
}
```

执行结果:

```
course1: Course{name='英语', score=12.0}
course2: Course{name='英语', score=12.0}
```

在默认情况下， 对于一个实例的多个引用，为了节省空间，只会写入一次，后面会追加几个字节代表某个实例的引用。

- 子类实现序列化，父类不实现序列化/ 对象引用

```
public class Person {

    private String name;
    private String sax;

//    public Person() {
//    }

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

public class Student1 extends Person implements Serializable {

    private static final long serialVersionUID = -2100492893943893602L;
```

```
    private Integer age;
    private List<Course> courses;

    public Student1(String name, String sax, Integer age) {
        super(name,sax);
       ...
    }
...

    public static void main(String ... args) throws Exception{
            //TODO:
        Student1 student = new Student1("Zero", "男", 18);
        student.addCourse(new Course("语文", 90.2f));
        //序列化
        byte[] bytes = SerializeableUtils.serialize(student);
        System.out.println(Arrays.toString(bytes));

        //反序列化
        //在readObject时抛出java.io.NotSerializableException异常。
        //需要Person添加一个无参数构造器
        Student1 student1 = SerializeableUtils.deserialize(bytes);
        System.out.println("Student: " + student1);
    }
```

在readObject时抛出java.io.NotSerializableException异常。

- 类的演化

```
//反序列化目标类多一个字段(height)
public class Student implements Serializable {

    private static final long serialVersionUID = -2100492893943893602L;
    private String name;
    private String sax;
    private Integer age;
    private List<Course> courses;

...

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", sax='" + sax + '\'' +
                ", age=" + age +
   //             ", height=" + height +
                ", courses=" + courses +
                '}';
    }

    //private float height;
    public static void main(String ... args)throws Exception{
            //TODO:
        String path = System.getProperty("user.dir") +"/a.out";
//        Student student = new Student("Zero", "男", 18);
//        student.addCourse(new Course("语文", 90.2f));
//        //序列化
```

```
//        SerializeableUtils.saveObject(student,path);


        //反序列化
        Student student1 = SerializeableUtils.readObject(path);
        System.out.println("Student: " + student1);
    }
}

```

执行结果：

```
//序列化的时候
Student: Zero 男 18
Course: 语文 90.2
//反序列化的时候 添加一个float height
Student: Student{name='Zero', sax='男', age=18, height=0.0, courses=
[Course{name='语文', score=90.2}]}
```

可以看出反序列化之后，并没有报错，只是height实赋成了默认值。类似的其它对象也会赋值为默认值。
还有 相反，如果写入的多一个字段，读出的少一个字段，也是不会报错的
其它演化，比如更改类型等，这种演化本身就有问题，没必再探讨

- 枚举类型

```
public enum Num {
    TWO, ONE, THREE;

    public void printValues() {
        System.out.println(ONE + " ONE.ordinal " + ONE.ordinal());
        System.out.println(TWO + " TWO.ordinal " + TWO.ordinal());
        System.out.println(THREE + " THREE.ordinal " + THREE.ordinal());
    }

    public static void testSerializable() throws Exception {
        File file = new File("p.dat");
//        ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(file));
//        oos.writeObject(Num.ONE);
//        oos.close();
        Num.ONE.printValues();
        System.out.println("==========反序列化后=======");

        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(file));
        Num s1 = (Num) ois.readObject();
        s1.printValues();
        ois.close();
    }

    public static void main(String... args) throws Exception {
        //TODO:
        testSerializable();
    }
}
```

执行结果:

```
        ONE ONE.ordinal 1
        TWO TWO.ordinal 0
        THREE THREE.ordinal 2
       =========反序列化后=======
      //调换(ONE,TWO)的位置: TWO, ONE, THREE; ->ONE, TWO, THREE;
        ONE ONE.ordinal 0
        TWO TWO.ordinal 1
        THREE THREE.ordinal 2
```

可以看到ONE的值变成了0.
事实上序列化Enum对象时，并不会保存元素的值，只会保存元素的name。这样，在不依赖元素值的前提下，ENUM对象如何更改都会保持兼容性。

重写readObject,writeObject

> “只有当你自行设计的自定义序列化形式与默认的序列化形式基本相同时，才能接受默认的序列化> 形式”.“当一个对象的物理表示方法与它的逻辑数据内容有实质性差别时，使用默认序列化形式有> > N种缺陷”.其实从effective java的角度来讲，是强烈建议我们重写的，这样有助于我们更好地把控> > 序列化过程，防范未知风险

```
public class Course3 implements Serializable {

    private static final long serialVersionUID = 667279791530738499L;
    private String name;
    private float score;
...

    private void readObject(ObjectInputStream inputStream) throws
ClassNotFoundException, IOException {
        System.out.println("readObject");
        inputStream.defaultReadObject();
        name = (String)inputStream.readObject();
        score = inputStream.readFloat();
    }

    private void writeObject(ObjectOutputStream outputStream) throws
IOException {
        System.out.println("writeObject");
        outputStream.defaultWriteObject();
        outputStream.writeObject(name);
        outputStream.writeFloat(score);
    }

    private Object readResolve() {
        System.out.println("readResolve");
        return new Course3(name, 85f);
    }

    private Object writeReplace(){
        System.out.println("writeReplace");
        return new Course3(name +"replace",score);
    }
```

```
...

    public static void main(String... args) throws Exception {
        //TODO:
        Course3 course = new Course3("英语", 12f);
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(out);
        oos.writeObject(course);
        byte[] bs = out.toByteArray();
        oos.close();

        ObjectInputStream ois = new ObjectInputStream(new
ByteArrayInputStream(bs));
        Course3 course1 = (Course3) ois.readObject();
        System.out.println("course1: " + course1);

    }


}
```

执行结果:

```
Course: 英语 12.0
writeReplace
Course: 英语replace 12.0
writeObject
readObject
readResolve
Course: 英语replace 85.0
course1: Course{name='英语replace', score=85.0}
```

- writeReplace 先于writeObject
- readResolve后于readObject

单例模式的序列化问题/反射问题

```
public class SingleTest {

    static final String CurPath = System.getProperty("user.dir");

    public static void main(String ... args) throws Exception {
            //TODO:
        Single instance = Single.getInstance();
        System.out.println(instance.hashCode());
        System.out.println(copyInstance(instance).hashCode());

        System.out.println("================反射====================");
        //使用反射方式直接调用私有构造器
        Class<Single> clazz =
(Class<Single>)Class.forName("com.zero.serializabledemo.serializable.Single"
);
        Constructor<Single> con = clazz.getDeclaredConstructor(null);
        con.setAccessible(true);//绕过权限管理，即在true的情况下，可以通过构造函数
新建对象
        Single instance1 = con.newInstance();
```

```
        Single instance2 = con.newInstance();
        System.out.println(instance1.hashCode());
        System.out.println(instance2.hashCode());

    }

    private static Single copyInstance(Single instance) throws Exception{
        //序列化会导致单例失效
        FileOutputStream fos = new FileOutputStream(CurPath+"/a.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(instance);
        ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(CurPath+"/a.txt"));
        Single single2 = (Single)ois.readObject();
        oos.close();
        ois.close();
        return single2;
    }
}

class Single implements Serializable {
    private static final long serialVersionUID = 1L;
    private static boolean flag = false;
    private Single(){
        synchronized (Single.class) {
            if (!flag) {
//                flag = true;
            } else {
                throw new RuntimeException("单例模式被侵犯！");
            }

        }
    }

    private static Single single;

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

        }
        return single;
    }
//如果不重写readResolve,会导致单例模式在序列化->反序列化后失败
//    private Object readResolve() {
//        return single;
//    }
}

```

Parcelable接口

介绍Parcelable不得不先提一下Serializable接口,Serializable是Java为我们提供的一个标准化的序列化接口,那什么是序列化呢? ---- 简单来说就是将对象转换为可以传输的二进制流(二进制序列)的过程,这样我们就可以通过序列化,转化为可以在网络传输或者保存到本地的流(序列),从而进行传输数据 ,那反序列化就是从二进制流(序列)转化为对象的过程.

Parcelable是Android为我们提供的序列化的接口,Parcelable相对于Serializable的使用相对复杂一些,但Parcelable的效率相对Serializable也高很多,这一直是Google工程师引以为傲的,有时间的可以看一下Parcelable和Serializable的效率对比 Parcelable vs Serializable 号称快10倍的效率

Parcelable是Android SDK提供的，它是基于内存的，由于内存读写速度高于硬盘，因此Android中的跨进程对象的传递一般使用Parcelable

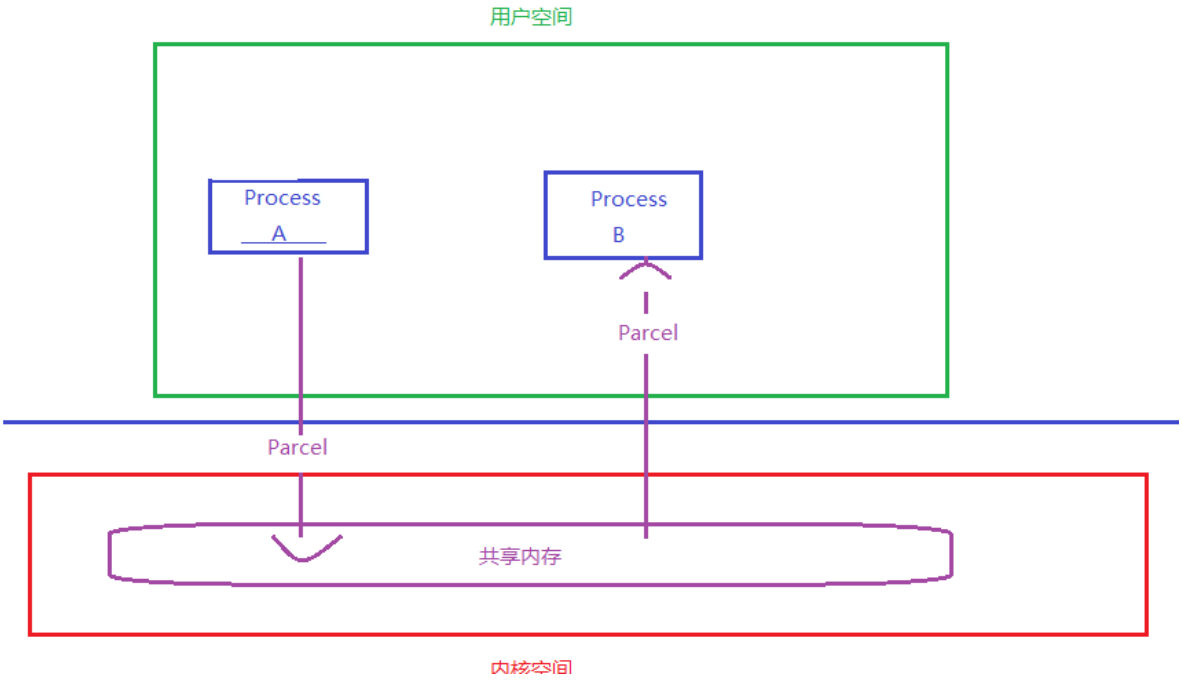
Parcelable入门

```
public class Course implements Parcelable {

    private String name;
    private float score;
...

  /**
     * 描述当前 Parcelable 实例的对象类型
     * 比如说，如果对象中有文件描述符，这个方法就会返回上面的CONTENTS_FILE_DESCRIPTOR
     * 其他情况会返回一个位掩码
     * @return
     */
    @Override
    public int describeContents() {
        return 0;
    }

 /**
     * 将对象转换成一个 Parcel 对象
     * @param dest 表示要写入的 Parcel 对象
     * @param flags 示这个对象将如何写入
     */
    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeString(this.name);
        dest.writeFloat(this.score);
    }

    protected Course(Parcel in) {
        this.name = in.readString();
        this.score = in.readFloat();
    }

/**
     * 实现类必须有一个 Creator 属性，用于反序列化，将 Parcel 对象转换为 Parcelable
     * @param <T>
     */
    public static final Parcelable.Creator<Course> CREATOR = new Parcelable.Creator<Course>() {

//反序列化的方法，将Parcel还原成Java对象
        @Override
```

```
        public Course createFromParcel(Parcel source) {
            return new Course(source);
        }

//提供给外部类反序列化这个数组使用。
        @Override
        public Course[] newArray(int size) {
            return new Course[size];
        }
    };
}
```

Parcel的简介

在介绍之前我们需要先了解Parcel是什么?Parcel翻译过来是打包的意思,其实就是包装了我们需要传输的数据,然后在Binder中传输,也就是用于跨进程传输数据
简单来说，Parcel提供了一套机制，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,下图是这个过程的模型。



Parcel可以包含原始数据类型（用各种对应的方法写入，比如writeInt(),writeFloat()等），可以包含Parcelable对象，它还包含了一个活动的IBinder对象的引用，这个引用导致另一端接收到一个指向这个IBinder的代理IBinder。

Parcelable通过Parcel实现了read和write的方法,从而实现序列化和反序列化,

Parcelable与Serializable的性能比较

Serializable性能分析

Serializable是Java中的序列化接口，其使用起来简单但开销较大（因为Serializable在序列化过程中使用了反射机制，故而会产生大量的临时变量，从而导致频繁的GC），并且在读写数据过程中，它是通过IO流的形式将数据写入到硬盘或者传输到网络上。

Parcelable性能分析

Parcelable则是以IBinder作为信息载体，在内存上开销比较小，因此在内存之间进行数据传递时，推荐使用Parcelable,而Parcelable对数据进行持久化或者网络传输时操作复杂，一般这个时候推荐使用Serializable。

性能比较总结描述

首先Parcelable的性能要强于Serializable的原因我需要简单的阐述一下

- 在内存的使用中,前者在性能方面要强于后者
- 后者在序列化操作的时候会产生大量的临时变量,(原因是使用了反射机制)从而导致GC的频繁调用,因此在性能上会稍微逊色
- Parcelable是以Ibinder作为信息载体的.在内存上的开销比较小,因此在内存之间进行数据传递的时候,Android推荐使用Parcelable,既然是内存方面比价有优势,那么自然就要优先选择.
- 在读写数据的时候,Parcelable是在内存中直接进行读写,而Serializable是通过使用IO流的形式将数据读写入在硬盘上.
  但是：虽然Parcelable的性能要强于Serializable,但是仍然有特殊的情况需要使用Serializable,而不去使用Parcelable,因为Parcelable无法将数据进行持久化,因此在将数据保存在磁盘的时候,仍然需要使用后者,因为前者无法很好的将数据进行持久化.(原因是在不同的Android版本当中,Parcelable可能会不同,因此数据的持久化方面仍然是使用Serializable)

性能测试方法分析

-
  - 通过将一个对象放到一个bundle里面然后调用Bundle#writeToParcel(Parcel, int)方法来模拟传递对象给一个activity的过程，然后再把这个对象取出来。
- 在一个循环里面运行1000 次。
- 两种方法分别运行10次来减少内存整理，cpu被其他应用占用等情况的干扰。
- 参与测试的对象就是上面的相关代码
- 在多种Android软硬件环境上进行测试

两种如何选择

- 在使用内存方面，Parcelable比Serializable性能高，所以推荐使用Parcelable。
- Serializable在序列化的时候会产生大量的临时变量，从而引起频繁的GC。
- Parcelable不能使用在要将数据存储在磁盘上的情况，因为Parcelable不能很好的保证数据的持续性，在外界有变化的情况下，建议使用Serializable

SQLite 与 SharedPreferences

- SQLite主要用于存储复杂的关系型数据，Android支持原生支持SQLite数据库相关操作（SQLiteOpenHelper）,不过由于原生API接口并不友好，所以产生了不少封装了SQLite的ORM框架。
- SharedPreferences是Android平台上提供的一个轻量级存储API，一般用于存储常用的配置信息，其本质是一个键值对存储，支持常用的数据类型如boolean、float、int、long以及String的存储和读取。

最后附带的几个面试相关的问题

- Android里面为什么要设计出Bundle而不是直接用Map结构

  Bundle内部是由ArrayMap实现的，ArrayMap的内部实现是两个数组，一个int数组是存储对象数据对应下标，一个对象数组保存key和value，内部使用二分法对key进行排序，所以在添加、删除、查找数据的时候，都会使用二分法查找，只适合于小数据量操作，如果在数据量比较大的情况下，那么它的性能将退化。而HashMap内部则是数组+链表结构，所以在数据量较少的时候，

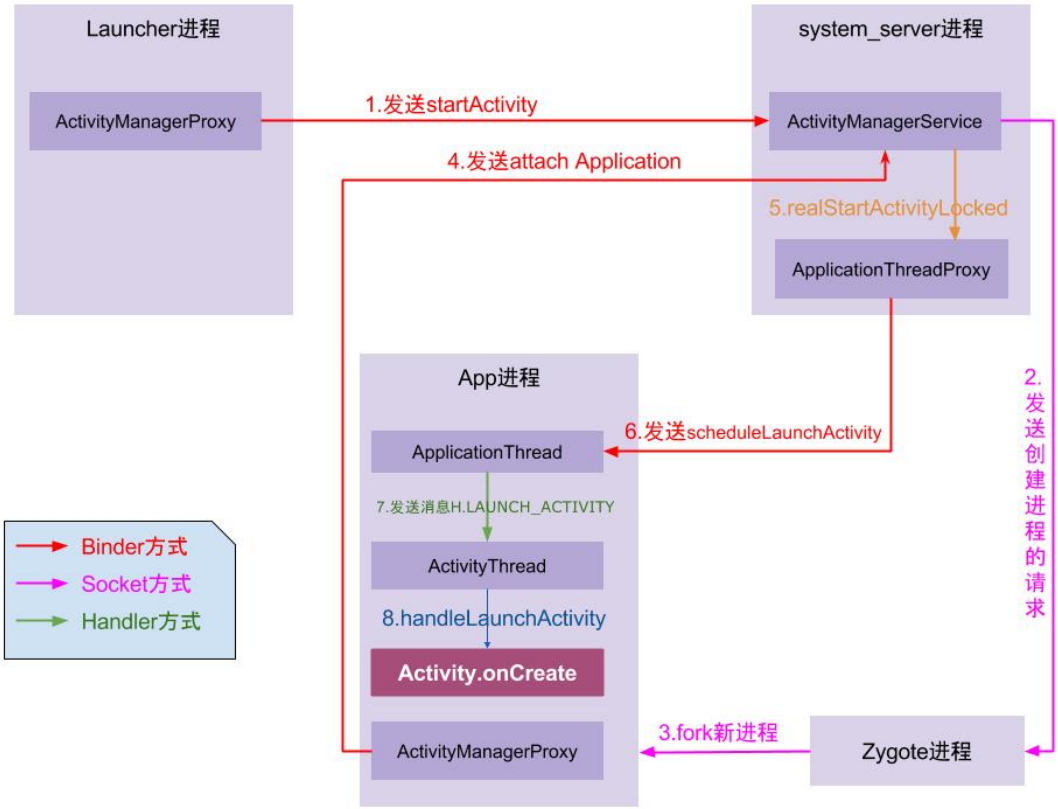
HashMap的Entry Array比ArrayMap占用更多的内存。因为使用Bundle的场景大多数为小数据量，我没见过在两个Activity之间传递10个以上数据的场景，所以相比之下，在这种情况下使用ArrayMap保存数据，在操作速度和内存占用上都具有优势，因此使用Bundle来传递数据，可以保证更快的速度和更少的内存占用。
另外一个原因，则是在Android中如果使用Intent来携带数据的话，需要数据是基本类型或者是可序列化类型，HashMap使用Serializable进行序列化，而Bundle则是使用Parcelable进行序列化。而在Android平台中，更推荐使用Parcelable实现序列化，虽然写法复杂，但是开销更小，所以为了更加快速的进行数据的序列化和反序列化，系统封装了Bundle类，方便我们进行数据的传输。

- Android中Intent/Bundle的通信原理及大小限制

  Intent 中的 Bundle 是使用 Binder 机制进行数据传送的。能使用的 Binder 的缓冲区是有大小限制的（有些手机是 2 M），而一个进程默认有 16 个 Binder 线程，所以一个线程能占用的缓冲区就更小了（有人以前做过测试，大约一个线程可以占用 128 KB）。所以当你看到 The Binder transaction failed because it was too large 这类 TransactionTooLargeException 异常时，你应该知道怎么解决了

- 为何Intent不能直接在组件间传递对象而要通过序列化机制?

  Intent在启动其他组件时，会离开当前应用程序进程，进入ActivityManagerService进程（intent.prepareToLeaveProcess()），这也就意味着，Intent所携带的数据要能够在不同进程间传输。首先我们知道，Android是基于Linux系统，不同进程之间的java对象是无法传输，所以我们此处要对对象进行序列化，从而实现对象在 应用程序进程 和 ActivityManagerService进程 之间传输。
  而Parcel或者Serializable都可以将对象序列化，其中，Serializable使用方便，但性能不如Parcel容器，后者也是Android系统专门推出的用于进程间通信等的接口



参考

> http://gityuan.com/2016/03/12/start-activity/