21.序列化

21.1.什么是序列化,

Java序列化是指把Java对象转换为字节序列的过程

Java反序列化是指把字节序列恢复为Java对象的过程；

21.2.为什么需要使用序列化和反序列化

不同进程/程序间进行远程通信时，可以相互发送各种类型的数据，包括文本、图片、音频、视频等，而这些数据都会以二进制序列的形式在网络上传送。

当两个Java进程进行通信时，对于进程间的对象传送需要使用Java序列化与反序列化了。发送方需要把这个Java对象转换为字节序列，接收方则需要将字节序列中恢复出Java对象。

21.3. 序列化的有哪些好处

实现了数据的持久化，通过序列化可以把数据永久地保存到硬盘上（如：存储在文件里），实现永久保存对象。

利用序列化实现远程通信，即：能够在网络上传输对象。

21.4. Serializable 和 Parcelable 的区别

Serializable原理(https://juejin.im/post/6844904049997774856)

Serializable接口没有方法和属性，只是一个识别类可被序列化的标志。

Serializable是通过I/O读写存储在磁盘上的, 通过反射解析出对象描述、属性的描述 以HandleTable来缓存解析信息,之后解析成二进制，存储、传输。

Parcel原理(https://www.wanandroid.com/wenda/show/9002)

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 ,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,

它的各种writeXXX方法，在native层都是会调用Parcel.cpp的write方法

Serializable 和 Parcelable 的区别

存储媒介的不同(https://www.jianshu.com/p/1b362e374354)

```
  Serializable 使用 I/O 读写存储在硬盘上，而 Parcelable 是直接 在内存中读写。很明显，内存的读写速度通常大于 IO 读写，所以在 Android 中传递数据优先选择 Parcelable。
```

效率不同

Serializable 会使用反射，序列化和反序列化过程需要大量 I/O 操作，

Parcelable 自已实现封送和解封（marshalled &unmarshalled）操作不需要用反射，数据也存放在 Native 内存中，效率要快很多。

21.5.什么是serialVersionUID

https://cloud.tencent.com/developer/article/1524781

序列化是将对象的状态信息转换为可存储或传输的形式的过程。我们都知道，Java对象是保存在JVM的堆内存中的，也就是说，如果JVM堆不存在了，那么对象也就跟着消失了。

而序列化提供了一种方案，可以让你在即使JVM停机的情况下也能把对象保存下来的方案。就像我们平时用的U盘一样。把Java对象序列化成可存储或传输的形式（如二进制流），比如保存在文件中。这样，当再次需要这个对象的时候，从文件中读取出二进制流，再从二进制流中反序列化出对象。

虚拟机是否允许反序列化，不仅取决于类路径和功能代码是否一致，一个非常重要的一点是两个类的序列化 ID 是否一致，这个所谓的序列化ID，就是我们在代码中定义的serialVersionUID。

21.6.为什么还要显示指定serialVersionUID的值?

如果不显示指定serialVersionUID, JVM在序列化时会根据属性自动生成一个serialVersionUID, 然后与属性一起序列化, 再进行持久化或网络传输. 在反序列化时, JVM会再根据属性自动生成一个新版serialVersionUID, 然后将这个新版serialVersionUID与序列化时生成的旧版serialVersionUID进行比较, 如果相同则反序列化成功, 否则报错.

如果显示指定了serialVersionUID, JVM在序列化和反序列化时仍然都会生成一个serialVersionUID, 但值为我们显示指定的值, 这样在反序列化时新旧版本的serialVersionUID就一致了.

在实际开发中, 不显示指定serialVersionUID的情况会导致什么问题? 如果我们的类写完后不再修改, 那当然不会有问题, 但这在实际开发中是不可能的, 我们的类会不断迭代, 一旦类被修改了, 那旧对象反序列化就会报错. 所以在实际开发中, 我们都会显示指定一个serialVersionUID, 值是多少无所谓, 只要不变就行.

22.Art & Dalvik 及其区别

22.1Art & Dalvik 及其区别

https://paul.pub/android-dalvik-vm/?spm=a2c6h.12873639.0.0.35ec6884Lt7nzq

https://paul.pub/android-art-vm/?spm=a2c6h.12873639.0.0.35ec6884Lt7nzq

Dalvik，ART是Android的两种运行环境，也可以叫做Android虚拟机 JIT，AOT是Android虚拟机采用的两种不同的编译策略

在Dalvik虚拟机上，APK中的Dex文件在安装时会被优化成odex文件，在运行时，会被JIT编译器编译成native代码。

在ART虚拟机上安装时，Dex文件会直接由dex2oat工具翻译成oat格式的文件，oat文件中既包含了dex文件中原先的内容，也包含了已经编译好的native代码。

22.1.1.Dalvik

1.原理

一个应用首先经过DX工具将class文件转换成Dalvik虚拟机可以执行的dex文件，然后由类加载器加载原生类和Java类。Dalvik虚拟机负责解释器根据指令集对Dalvik字节码进行释dex文件为机器码。

JIT编译器

Dalvik负责将dex翻译为机器码交由系统调用，有一个缺陷，每次执行代码，都需要Dalvik将操作码代码翻译为机器对应的微处理器指令，然后交给底层系统处理，运行效率很低。

JIT编译器，当App运行时，每当遇到一个新类，JIT编译器就会对这个类进行即时编译，经过编译后的代码，会被优化成相当精简的原生型指令码（即native code），这样在下次执行到相同逻辑的时候，速度就会更快。

2.Dalvik的启动流程

Dalvik进程管理是依赖于linux的进程体系结构的，如要为应用程序创建一个进程，它会使用linux的fork机制来复制一个进程。

22.1.2.ART

1.原理

JIT是运行时编译，这样可以对执行次数频繁的dex代码进行编译和优化，减少以后使用时的翻译时间， 但将dex翻译为本地机器码也要占用时间，所以Google在4.4之后推出了ART，用来替换Dalvik。

ART的策略与Dalvik不同，在ART 环境中，应用在第一次安装的时候，字节码就会预先编译成机器码，使其成为真正的本地应用。之后打开App的时候，不需要额外的翻译工作，直接使用本地机器码运行，因此运行速度提高。

AOT

AOT 是静态编译，应用在安装的时候会启动 dex2oat 过程把 dex预编译成 ELF 文件，每次运行程序的时候不用重新编译。