如梦朦胧

九月份的时候有了换工作的躁动,然后投了某度的Android岗位,本以为像我这种非211、985没工作经验的渣渣只能被直接pass,结果却意外的收到了电话,真是受宠若惊.经过电面,技术三面,然后就是等通知到最后拿到了OFFER,如梦一般,真是挺激动的.

面试的准备

当收到HR的面试的通知还是很懵逼的,因为感觉自己突然啥都不会了,迅速镇定下来,去网上找了一下某度的面试题,但是发现都只有提问了什么并没有对所提问题的解答,那只能自力更生,像做试卷一样,一遍总结一遍温顾.其实大多都是平时开发中用到的,只是我们没有总结过,被问起来的时候回答的难免会有点捉襟见肘,不能回答的很全面.下面为我个人总(bai)结(du)的,希望对你能有所帮助,但毕竟能力有限,有写的不对的地方,还望轻喷.虽然喷我我也不会改的.

因为本文篇幅较长建议收藏,在用到时候找出来看一眼.有一些知识点可能没涉及到,以后会加以补足.因为面试无非是考察你对技术的理解和总结,所以本篇的每个点总结的比较精简,只是让你大概的说出来,有的部分是需要能够画出原理图并进行解释说明,这个要在工作中多积累.

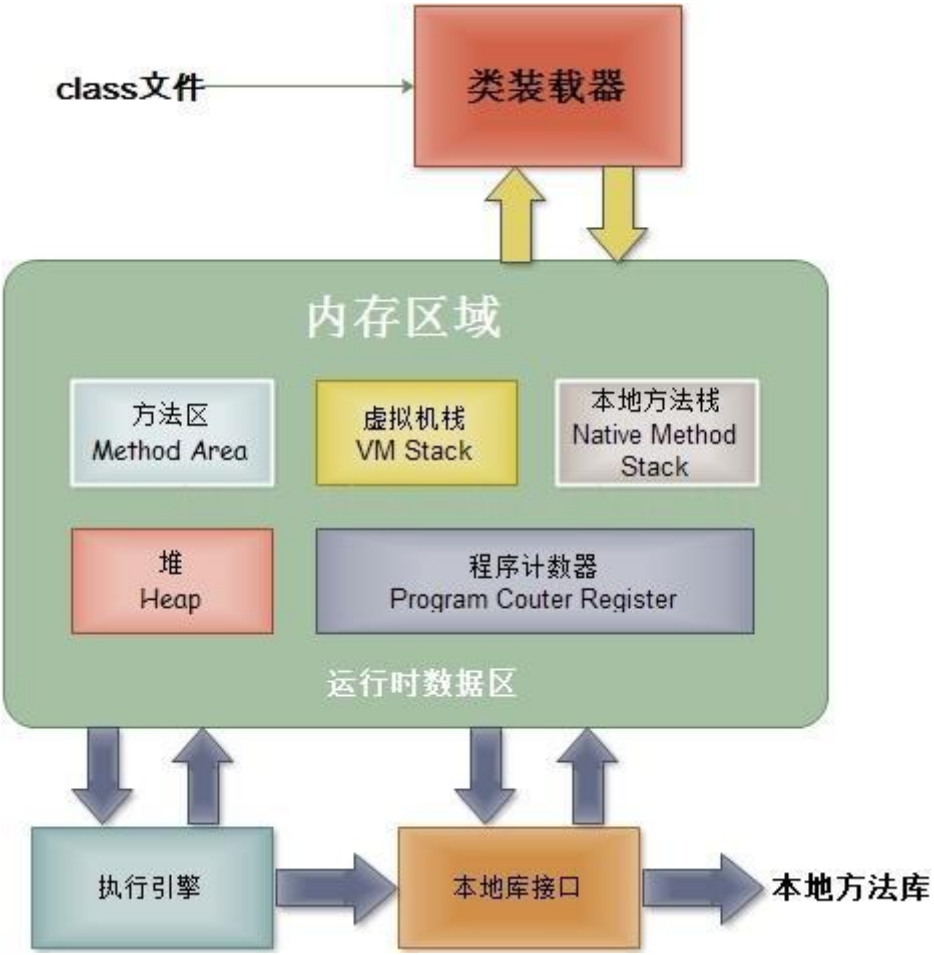
JAVA

一. 类的加载过程，Person person = new Person();为例进行说明。

1. 因为new用到了Person.class，所以会先找到Person.class文件，并加载到内存中;
2. 执行该类中的static代码块，如果有的话，给Person.class类进行初始化;
3. 在堆内存中开辟空间分配内存地址;
4. 在堆内存中建立对象的特有属性，并进行默认初始化;
5. 对属性进行显示初始化;
6. 对对象进行构造代码块初始化;
7. 对对象进行与之对应的构造函数进行初始化;
8. 将内存地址付给栈内存中的p变量

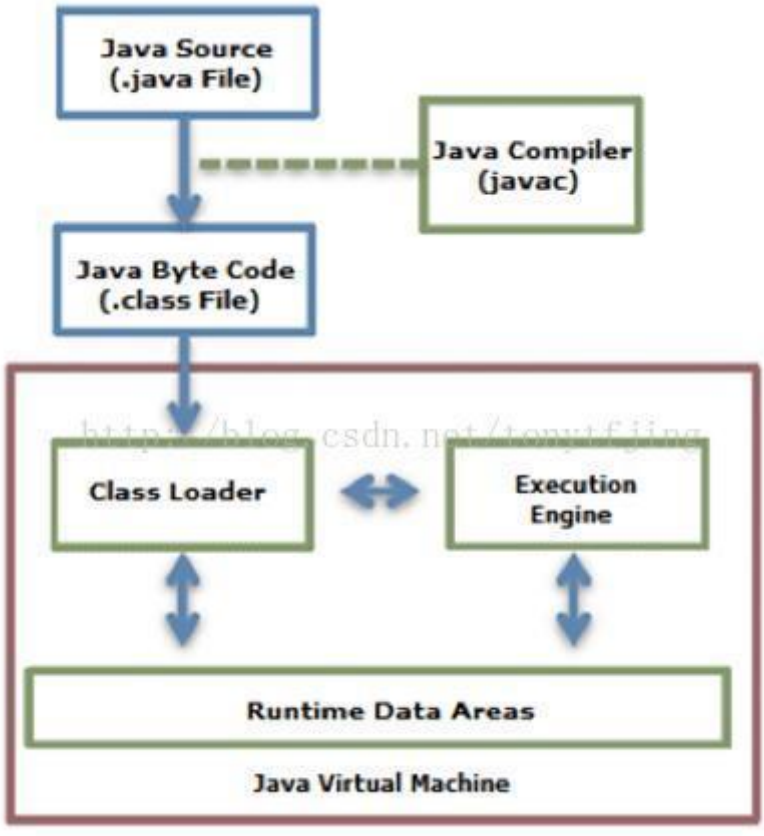
二. JVM相关知识，GC机制。

JVM基本构成



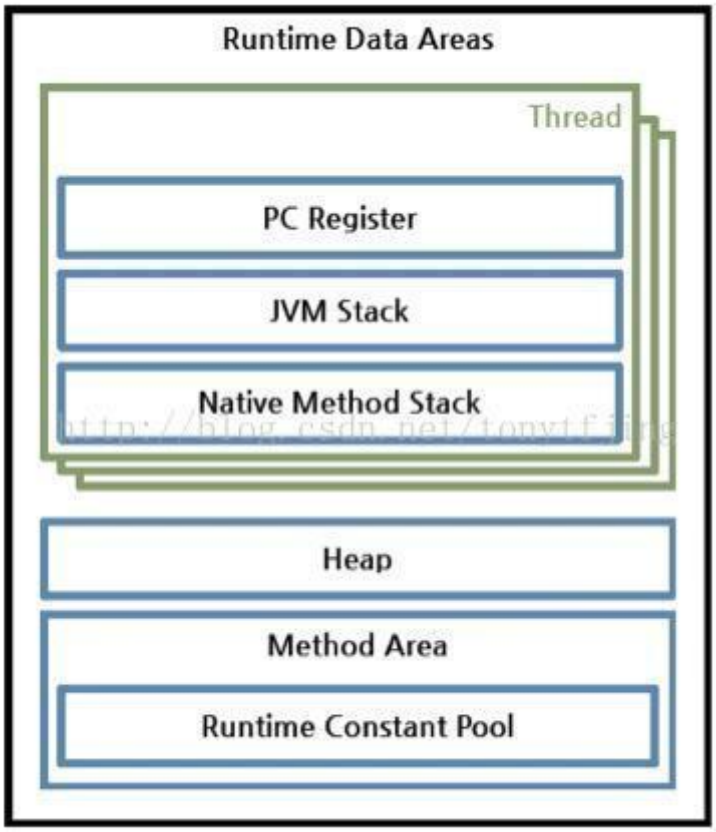
从上图可知，JVM主要包括四个部分：

1、类加载器（ClassLoader）:在JVM启动时或者在类运行时将需要的class加载到JVM中。（下图表示了从java源文件到JVM的整个过程，可配合理解。



2、执行引擎：负责执行class文件中包含的字节码指令；

3、内存区（也叫运行时数据区）：是在JVM运行的时候操作所分配的内存区。运行时内存区主要可以划分为5个区域，如图：



- 方法区(Method Area)：用于存储类结构信息的地方，包括常量池、静态变量、构造函数等。虽然JVM规范把方法区描述为堆的一个逻辑部分， 但它却有个别名non-heap（非堆），所以大家不要搞混淆了。方法区还包含一个运行时常量池。
- java堆(Heap)：存储java实例或者对象的地方。这块是GC的主要区域。从存储的内容我们可以很容易知道，方法区和堆是被所有java线程共享的。
- java栈(Stack)：java栈总是和线程关联在一起，每当创建一个线程时，JVM就会为这个线程创建一个对应的java栈。在这个java栈中又会包含多个栈帧，每运行一个方法就创建一个栈帧，用于存储局部变量表、操作栈、方法返回值等。每一个方法从调用直至执行完成的过程，就对应一个栈帧在java栈中入栈到出栈的过程。所以java栈是现成私有的。
- 程序计数器(PC Register)：用于保存当前线程执行的内存地址。由于JVM程序是多线程执行的（线程轮流切换），所以为了保证线程切换回来后，还能恢复到原先状态，就需要一个独立的计数器，记录之前中断的地方，可见程序计数器也是线程私有的。
- 本地方法栈(Native Method Stack)：和java栈的作用差不多，只不过是为JVM使用到的native方法服务的。

4、本地方法接口：主要是调用C或C++实现的本地方法及返回结果。

GC机制

垃圾收集器一般必须完成两件事：检测出垃圾；回收垃圾。怎么检测出垃圾？一般有以下几种方法：

引用计数法：

给一个对象添加引用计数器，每当有个地方引用它，计数器就加1；引用失效就减1。好了，问题来了，如果我有两个对象A和B，互相引用，除此之外，没有其他任何对象引用它们，实际上这两个对象已经无法访问，即是我们说的垃圾对象。但是互相引用，计数不为0，导致无法回收，所以还有另一种方法：

可达性分析算法：

以根集对象为起始点进行搜索，如果有对象不可达的话，即是垃圾对象。这里的根集一般包括java栈中引用的对象、方法区常良池中引用的对象、本地方法中引用的对象等。

总之，JVM在做垃圾回收的时候，会检查堆中的所有对象是否会被这些根集对象引用，不能够被引用的对象就会被垃圾收集器回收。一般回收算法也有如下几种：

1. 标记-清除（Mark-sweep）
2. 复制（Copying
3. 标记-整理（Mark-Compact）
4. 分代收集算法

三. 类的加载器，双亲机制，Android的类加载器。

类的加载器

大家都知道，当我们写好一个Java程序之后，不是管是CS还是BS应用，都是由若干个.class文件组织而成的一个完整的Java应用程序，当程序在运行时，即会调用该程序的一个入口函数来调用系统的相关功能，而这些功能都被封装在不同的class文件当中，所以经常要从这个class文件中要调用另外一个class文件中的方法，如果另外一个文件不存在的，则会引发系统异常。而程序在启动的时候，并不会一次性加载程序所要用的所有class文件，而是根据程序的需要，通过Java的类加载机制（ClassLoader）来动态加载某个class文件到内存当中的，从而只有class文件被载入到了内存之后，才能被其它class所引用。所以ClassLoader就是用来动态加载class文件到内存当中用的。

双亲机制

1、原理介绍

ClassLoader使用的是双亲委托模型来搜索类的，每个ClassLoader实例都有一个父类加载器的引用（不是继承的关系，是一个包含的关系），虚拟机内置的类加载器（Bootstrap ClassLoader）本身没有父类加载器，但可以用作其它ClassLoader实例的的父类加载器。当一个ClassLoader实例需要加载某个类时，它会试图亲自搜索某个类之前，先把这个任务委托给它的父类加载器，这个过程是由上至下依次检查的，首先由最顶层的类加载器Bootstrap ClassLoader试图加载，如果没加载到，则把任务转交给Extension ClassLoader试图加载，如果也没加载到，则转交给App ClassLoader 进行加载，如果它也没有加载得到的话，则返回给委托的发起者，由它到指定的文件系统或网络等URL中加载该类。如果它们都没有加载到这个类时，则抛出ClassNotFoundException异常。否则将这个找到的类生成一个类的定义，并将它加载到内存当中，最后返回这个类在内存中的Class实例对象。

2、为什么要使用双亲委托这种模型呢?

因为这样可以避免重复加载，当父亲已经加载了该类的时候，就没有必要子ClassLoader再加载一次。考虑到安全因素，我们试想一下，如果不使用这种委托模式，那我们就可以随时使用自定义的String来动态替代java核心api中定义的类型，这样会存在非常大的安全隐患，而双亲委托的方式，就可以避免这种情况，因为String已经在启动时就被引导类加载器（Bootstrcp ClassLoader）加载，所以用户自定义的ClassLoader永远也无法加载一个自己写的String，除非你改变JDK中ClassLoader搜索类的默认算法。

3、但是JVM在搜索类的时候，又是如何判定两个class是相同的呢?

JVM在判定两个class是否相同时，不仅要判断两个类名是否相同，而且要判断是否由同一个类加载器实例加载的。只有两者同时满足的情况下，JVM才认为这两个class是相同的。就算两个class是同一份class字节码，如果被两个不同的ClassLoader实例所加载，JVM也会认为它们是两个不同class。比如网络上的一个Java类org.classloader.simple.NetClassLoaderSimple，javac编译之后生成字节码文件NetClassLoaderSimple.class，ClassLoaderA和ClassLoaderB这两个类加载器并读取了NetClassLoaderSimple.class文件，并分别定义出了java.lang.Class实例来表示这个类，对于JVM来说，它们是两个不同的实例对象，但它们确实是同一份字节码文件，如果试图将这个Class实例生成具体的对象进行转换时，就会抛运行时异常java.lang.ClassCaseException，提示这是两个不同的类型。

Android类加载器

对于Android而言，最终的apk文件包含的是dex类型的文件，dex文件是将class文件重新打包，打包的规则又不是简单地压缩，而是完全对class文件内部的各种函数表，变量表进行优化，产生一个新的文件，即dex文件。因此加载这种特殊的Class文件就需要特殊的类加载器DexClassLoader。

四. 集合框架，list，map，set都有哪些具体的实现类，区别都是什么?

1、List,Set都是继承自Collection接口，Map则不是;

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，重复元素会覆盖掉，（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，加入Set 的Object必须定义equals()方法;
另外list支持for循环，也就是通过下标来遍历，也可以用迭代器，但是set只能用迭代，因为他无序，无法用下标来取得想要的值）。

3、Set和List对比:
Set：检索元素效率低下，删除和插入效率高，插入和删除不会引起元素位置改变。
List：和数组类似，List可以动态增长，查找元素效率高，插入删除元素效率低，因为会引起其他元素位置改变。

4、Map适合储存键值对的数据。

5、线程安全集合类与非线程安全集合类
LinkedList、ArrayList、HashSet是非线程安全的，Vector是线程安全的;
HashMap是非线程安全的，HashTable是线程安全的;
StringBuilder是非线程安全的，StringBuffer是线程安全的。

下面是这些类具体的使用介绍:

ArrayList与LinkedList的区别和适用场景

Arraylist:

优点：ArrayList是实现了基于动态数组的数据结构,因为地址连续，一旦数据存储好了，查询操作效率会比较高（在内存里是连着放的）。
缺点：因为地址连续， ArrayList要移动数据,所以插入和删除操作效率比较低。

LinkedList:

优点：LinkedList基于链表的数据结构,地址是任意的，所以在开辟内存空间的时候不需要等一个连续的地址，对于新增和删除操作add和remove，LinedList比较占优势。LinkedList 适用于要头尾操作或插入指定位置的场景。
缺点：因为LinkedList要移动指针,所以查询操作性能比较低。
适用场景分析：当需要对数据进行对此访问的情况下选用ArrayList，当需要对数据进行多次增加删除修改时采用LinkedList。

ArrayList与Vector的区别和适用场景

ArrayList有三个构造方法:

```
public ArrayList(int initialCapacity)//构造一个具有指定初始容量的空列表。
public ArrayList()//构造一个初始容量为10的空列表。
public ArrayList(Collection<? extends E> c)//构造一个包含指定 collection 的元素的列表
```

Vector有四个构造方法:

```
public Vector()//使用指定的初始容量和等于零的容量增量构造一个空向量。
public Vector(int initialCapacity)//构造一个空向量，使其内部数据数组的大小，其标准容量增量为零。
public Vector(Collection<? extends E> c)//构造一个包含指定 collection 中的元素的向量

public Vector(int initialCapacity,int capacityIncrement)//使用指定的初始容量和容量增量构造一个空的向量
```

ArrayList和Vector都是用数组实现的，主要有这么三个区别：

1. Vector是多线程安全的，线程安全就是说多线程访问同一代码，不会产生不确定的结果。而ArrayList不是，这个可以从源码中看出，Vector类中的方法很多有synchronized进行修饰，这样就导致了Vector在效率上无法与ArrayList相比；
2. 两个都是采用的线性连续空间存储元素，但是当空间不足的时候，两个类的增加方式是不同。
3. Vector可以设置增长因子，而ArrayList不可以。
4. Vector是一种老的动态数组，是线程同步的，效率很低，一般不赞成使用。

适用场景：

1. Vector是线程同步的，所以它也是线程安全的，而ArrayList是线程异步的，是不安全的。如果不考虑到线程的安全因素，一般用ArrayList效率比较高。
2. 如果集合中的元素的数目大于目前集合数组的长度时，在集合中使用数据量比较大的数据，用Vector有一定的优势。

HashSet与Treeset的适用场景

1. TreeSet 是二叉树（红黑树的树据结构）实现的,Treeset中的数据是自动排好序的，不允许放入null值。
2. HashSet 是哈希表实现的,HashSet中的数据是无序的，可以放入null，但只能放入一个null，两者中的值都不能重复，就如数据库中唯一约束。
3. HashSet要求放入的对象必须实现HashCode()方法，放入的对象，是以hashcode码作为标识的，而具有相同内容的String对象，hashcode是一样，所以放入的内容不能重复。但是同一个类的对象可以放入不同的实例。

适用场景分析:

HashSet是基于Hash算法实现的，其性能通常都优于TreeSet。为快速查找而设计的Set，我们通常都应该使用HashSet，在我们需要排序的功能时，我们才使用TreeSet。

HashMap与TreeMap、HashTable的区别及适用场景

HashMap 非线程安全

HashMap：基于哈希表(散列表)实现。使用HashMap要求添加的键类明确定义了hashCode()和equals()[可以重写hashCode()和equals()]，为了优化HashMap空间的使用，您可以调优初始容量和负载因子。其中散列表的冲突处理主要分两种，一种是开放定址法，另一种是链表法。HashMap的实现中采用的是链表法。

TreeMap：非线程安全基于红黑树实现。TreeMap没有调优选项，因为该树总处于平衡状态。

适用场景分析：

HashMap和HashTable:HashMap去掉了HashTable的contains方法，但是加上了containsValue()和containsKey()方法。HashTable同步的，而HashMap是非同步的，效率上比HashTable要高。HashMap允许空键值，而HashTable不允许。

HashMap：适用于Map中插入、删除和定位元素。

Treemap：适用于按自然顺序或自定义顺序遍历键(key)。

(ps:其实我们工作的过程中对集合的使用是很频繁的,稍加注意并总结积累一下,在面试的时候应该会回答的很轻松)

五. concurrentHashmap原理，原子类。

ConcurrentHashMap作为一种线程安全且高效的哈希表的解决方案，尤其其中的"分段锁"的方案，相比HashTable的全表锁在性能上的提升非常之大.

六. volatile原理

在《Java并发编程：核心理论》一文中，我们已经提到过可见性、有序性及原子性问题，通常情况下我们可以通过Synchronized关键字来解决这些个问题，不过如果对Synchronized原理有了解的话，应该知道Synchronized是一个比较重量级的操作，对系统的性能有比较大的影响，所以，如果有其他解决方案，我们通常都避免使用Synchronized来解决问题。而volatile关键字就是Java中提供的另一种解决可见性和有序性问题的方案。对于原子性，需要强调一点，也是大家容易误解的一点：对volatile变量的单次读/写操作可以保证原子性的，如long和double类型变量，但是并不能保证i++这种操作的原子性，因为本质上i++是读、写两次操作。

七. 多线程的使用场景

使用多线程就一定效率高吗？ 有时候使用多线程并不是为了提高效率，而是使得CPU能够同时处理多个事件。

1. 为了不阻塞主线程,启动其他线程来做好事的事情,比如APP中耗时操作都不在UI中做.
2. 实现更快的应用程序,即主线程专门监听用户请求,子线程用来处理用户请求,以获得大的吞吐量.感觉这种情况下，多线程的效率未必高。 这种情况下的多线程是为了不必等待， 可以并行处理多条数据。比如JavaWeb的就是主线程专门监听用户的HTTP请求，然后启动子线程去处理用户的HTTP请求。
3. 某种虽然优先级很低的服务，但是却要不定时去做。 比如Jvm的垃圾回收。
4. 某种任务，虽然耗时，但是不耗CPU的操作时，开启多个线程，效率会有显著提高。比如读取文件，然后处理。 磁盘IO是个很耗费时间，但是不耗CPU计算的工作。 所以可以一个线程读取数据，一个线程处理数据。肯定比一个线程读取数据，然后处理效率高。 因为两个线程的时候充分利用了CPU等待磁盘IO的空闲时间。

八. JAVA常量池

Interger中的128(-128~127)

a.当数值范围为-128~127时：如果两个new出来Integer对象，即使值相同，通过“==”比较结果为false，但两个对象直接赋值，则通过“==”比较结果为“true，这一点与String非常相似。

b.当数值不在-128~127时，无论通过哪种方式，即使两个对象的值相等，通过“==”比较，其结果为false；

c.当一个Integer对象直接与一个int基本数据类型通过“==”比较，其结果与第一点相同；

d.Integer对象的hash值为数值本身；

为什么是-128-127?

在Integer类中有一个静态内部类IntegerCache，在IntegerCache类中有一个Integer数组，用以缓存当数值范围为-128~127时的Integer对象。

九. 简单介绍一下java中的泛型，泛型擦除以及相关的概念。

泛型是Java SE 1.5的新特性，泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。 Java语言引入泛型的好处是安全简单。

在Java SE 1.5之前，没有泛型的情况的下，通过对类型Object的引用来实现参数的“任意化”，“任意化”带来的缺点是要做显式的强制类型转换，而这种转换是要求开发者对实际参数类型可以预知的情况下进行的。对于强制类型转换错误的情况，编译器可能不提示错误，在运行的时候才出现异常，这是一个安全隐患。

泛型的好处是在编译的时候检查类型安全，并且所有的强制转换都是自动和隐式的，提高代码的重用率。

1. 泛型的类型参数只能是类类型（包括自定义类），不能是简单类型。
2. 同一种泛型可以对应多个版本（因为参数类型是不确定的），不同版本的泛型类实例是不兼容的。
3. 泛型的类型参数可以有多个。
4. 泛型的参数类型可以使用extends语句，例如。习惯上称为“有界类型”。
5. 泛型的参数类型还可以是通配符类型。例如Class<?> classType = Class.forName("java.lang.String");

泛型擦除以及相关的概念

Java中的泛型基本上都是在编译器这个层次来实现的。在生成的Java字节码中是不包含泛型中的类型信息的。使用泛型的时候加上的类型参数，会在编译器在编译的时候去掉。这个过程就称为类型擦除。

类型擦除引起的问题及解决方法

1. 先检查，在编译，以及检查编译的对象和引用传递的问题
2. 自动类型转换
3. 类型擦除与多态的冲突和解决方法
4. 泛型类型变量不能是基本数据类型
5. 运行时类型查询
6. 异常中使用泛型的问题
7. 数组（这个不属于类型擦除引起的问题）
8. 类型擦除后的冲突
9. 泛型在静态方法和静态类中的问题

Android

一. Handler机制

Android 的消息机制也就是 handler 机制,创建 handler 的时候会创建一个 looper (通过 looper.prepare() 来创建),looper 一般为主线程 looper.

handler 通过 send 发送消息 (sendMessage) ,当然 post 一系列方法最终也是通过 send 来实现的,在 send 方法中handler 会通过 enqueueMessage() 方法中的 enqueueMessage(msg,millis)向消息队列 MessageQueue 插入一条消息,同时会把本身的 handler 通过 msg.target = this 传入.

Looper 是一个死循环,不断的读取MessageQueue中的消息,loop 方法会调用 MessageQueue 的 next 方法来获取新的消息,next 操作是一个阻塞操作,当没有消息的时候 next 方法会一直阻塞,进而导致 loop 一直阻塞,当有消息的时候,Looper 就会处理消息 Looper 收到消息之后就开始处理消息: msg.target.dispatchMessage(msg),当然这里的 msg.target 就是上面传过来的发送这条消息的 handler 对象,这样 handler 发送的消息最终又交给他的dispatchMessage方法来处理了,这里不同的是,handler 的 dispatchMessage 方法是在创建 Handler时所使用的 Looper 中执行的,这样就成功的将代码逻辑切换到了主线程了.

Handler 处理消息的过程是:首先,检查Message 的 callback 是否为 null,不为 null 就通过 handleCallBack 来处理消息,Message 的 callback 是一个 Runnable 对象,实际上是 handler 的 post 方法所传递的 Runnable 参数.其次是检查 mCallback 是 否为 null,不为 null 就调用 mCallback 的 handleMessage 方法来处理消息.

二. View的绘制流程

View的绘制流程：OnMeasure()——>OnLayout()——>OnDraw()

各步骤的主要工作：

OnMeasure()：测量视图大小。从顶层父View到子View递归调用measure方法，measure方法又回调OnMeasure。

**OnLayout()：*确定View位置，进行页面布局。从顶层父View向子View的递归调用view.layout方法的过程，即父View根据上一步measure子View所得到的布局大小和布局参数，将子View放在合适的位置上。

OnDraw()：绘制视图:ViewRoot创建一个Canvas对象，然后调用OnDraw()。六个步骤：①、绘制视图的背景；②、保存画布的图层（Layer）；③、绘制View的内容；④、绘制View子视图，如果没有就不用；⑤、还原图层（Layer）；⑥、绘制滚动条。

三. 事件传递机制

1. Android事件分发机制的本质是要解决：点击事件由哪个对象发出，经过哪些对象，最终达到哪个对象并最终得到处理。这里的对象是指Activity、ViewGroup、View.
2. Android中事件分发顺序：Activity（Window） -> ViewGroup -> View.
3. 事件分发过程由dispatchTouchEvent() 、onInterceptTouchEvent()和onTouchEvent()三个方法协助完成

设置Button按钮来响应点击事件事件传递情况：（如下图）

布局如下:

- 最外层：Activiy A，包含两个子View：ViewGroup B、View C
- 中间层：ViewGroup B，包含一个子View：View C
- 最内层：View C

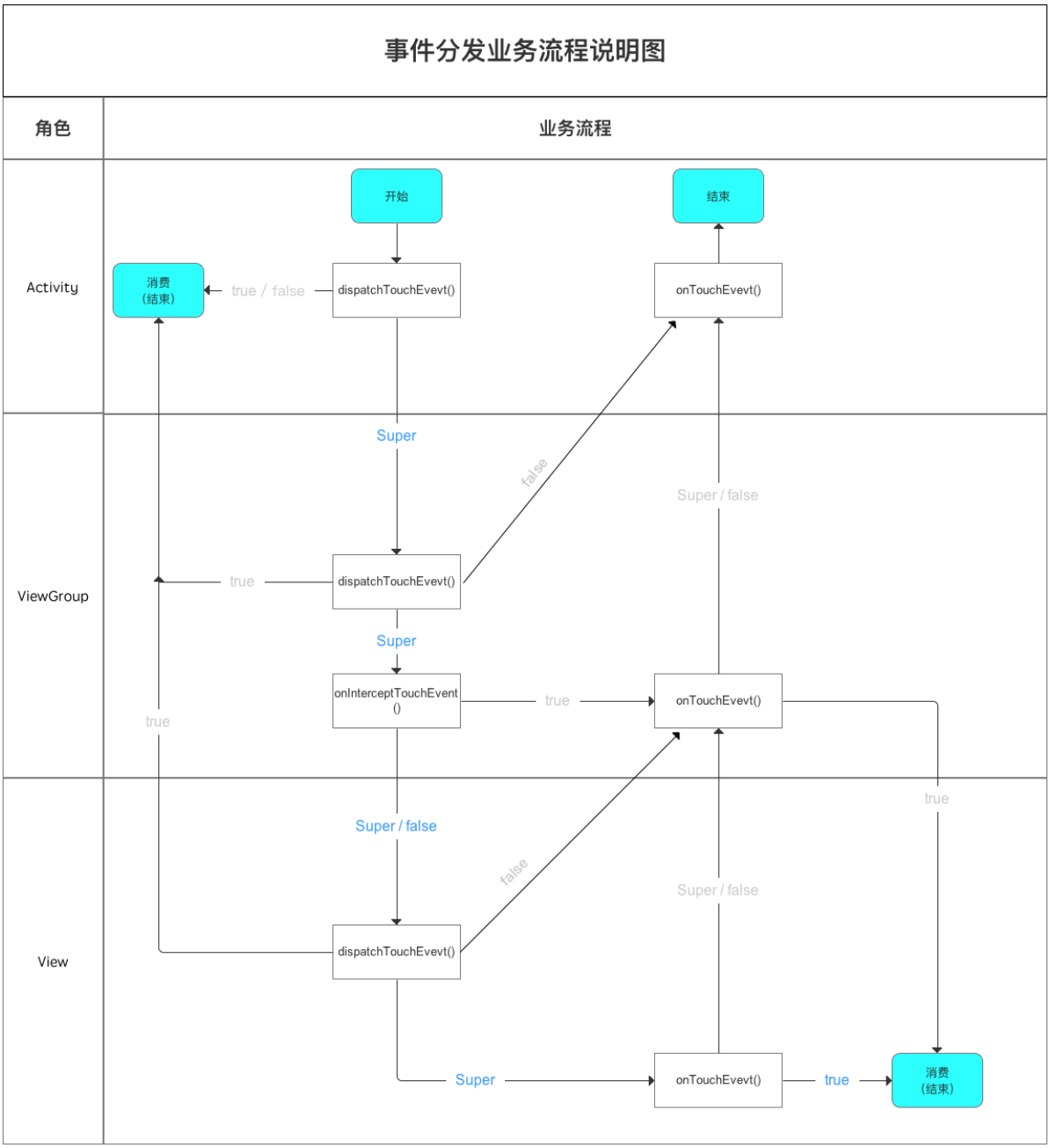
假设用户首先触摸到屏幕上View C上的某个点（如图中黄色区域），那么Action_DOWN事件就在该点产生，然后用户移动手指并最后离开屏幕。

按钮点击事件:

DOWN事件被传递给C的onTouchEvent方法，该方法返回true，表示处理这个事件;

因为C正在处理这个事件，那么DOWN事件将不再往上传递给B和A的onTouchEvent();

该事件列的其他事件（Move、Up）也将传递给C的onTouchEvent();



(记住这个图的传递顺序,面试的时候能够画出来,就很详细了)

四. Binder机制

1.了解Binder

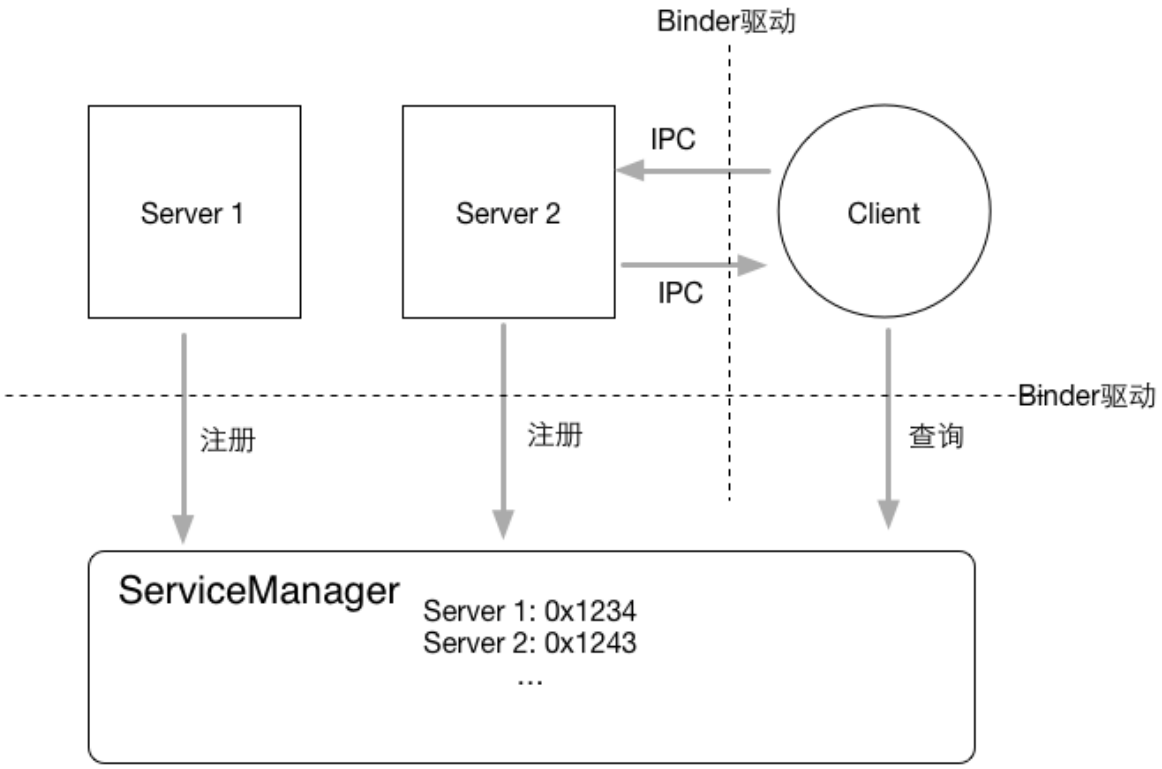
在Android系统中，每一个应用程序都运行在独立的进程中，这也保证了当其中一个程序出现异常而不会影响另一个应用程序的正常运转。在许多情况下，我们activity都会与各种系统的service打交道，很显然，我们写的程序中activity与系统service肯定不是同一个进程，但是它们之间是怎样实现通信的呢？所以Binder是android中一种实现进程间通信（IPC）的方式之一。

首先，Binder分为Client和Server两个进程。

注意，Client和Server是相对的。谁发消息，谁就是Client，谁接收消息，谁就是Server。

举个例子，两个进程A和B之间使用Binder通信，进程A发消息给进程B，那么这时候A是Binder Client，B是Binder Server；进程B发消息给进程A，那么这时候B是Binder Client，A是Binder Server——其实这么说虽然简单了，但还是不太严谨，我们先这么理解着。

其次，我们看下面这个图（摘自田维术的博客），基本说明白了Binder的组成解构：



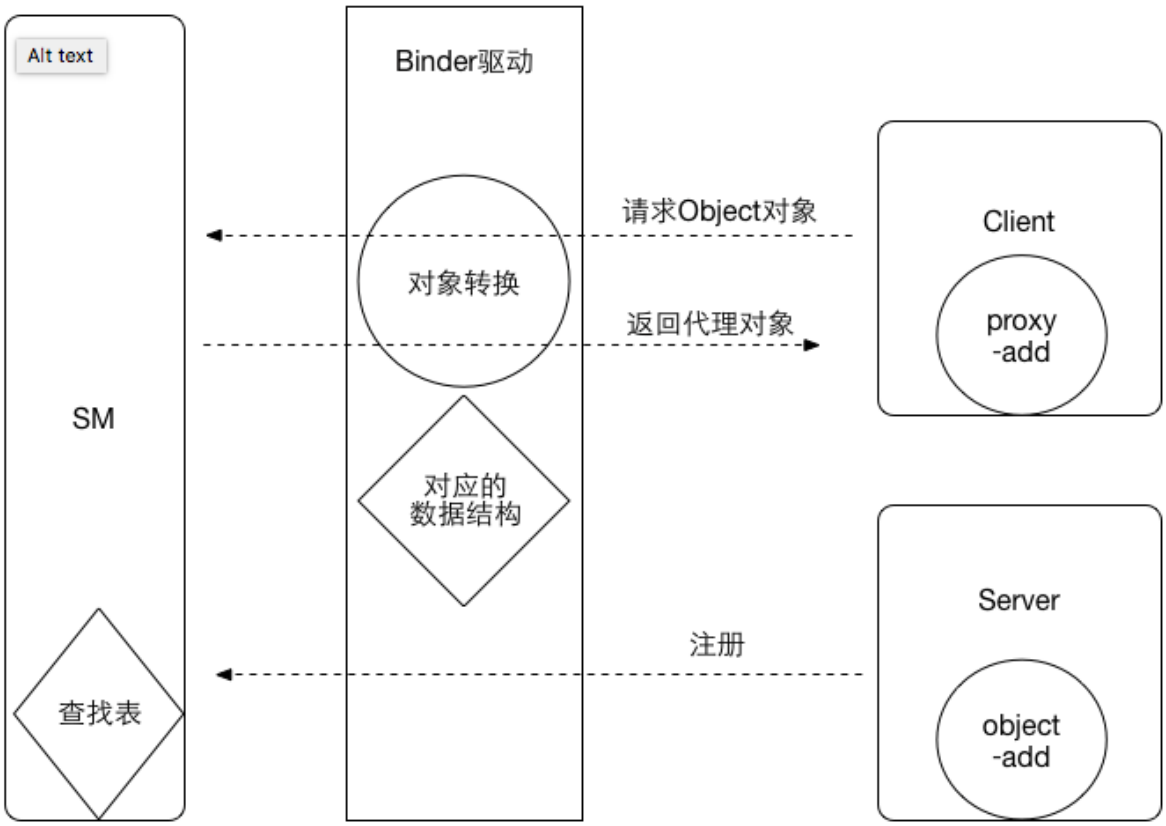
图中的IPC就是进程间通信的意思。

图中的ServiceManager，负责把Binder Server注册到一个容器中。

有人把ServiceManager比喻成电话局，存储着每个住宅的座机电话，还是很恰当的。张三给李四打电话，拨打电话号码，会先转接到电话局，电话局的接线员查到这个电话号码的地址，因为李四的电话号码之前在电话局注册过，所以就能拨通；没注册，就会提示该号码不存在。

对照着Android Binder机制，对着上面这图，张三就是Binder Client，李四就是Binder Server，电话局就是ServiceManager，电话局的接线员在这个过程中做了很多事情，对应着图中的Binder驱动.

接下来我们看Binder通信的过程，还是摘自田维术博客的一张图：



注：图中的SM也就是ServiceManager。

我们看到，Client想要直接调用Server的add方法，是不可以的，因为它们在不同的进程中，这时候就需要Binder来帮忙了。

首先是Server在SM这个容器中注册。

其次，Client想要调用Server的add方法，就需要先获取Server对象， 但是SM不会把真正的Server对象返回给Client，而是把Server的一个代理对象返回给Client，也就是Proxy。

然后，Client调用Proxy的add方法，SM会帮他去调用Server的add方法，并把结果返回给Client。

以上这3步，Binder驱动出了很多力，但我们不需要知道Binder驱动的底层实现，涉及到C++的代码了——把有限的时间去做更有意义的事情。

2.为什么android选用Binder来实现进程间通信?

1. 可靠性。在移动设备上，通常采用基于Client-Server的通信方式来实现互联网与设备间的内部通信。目前linux支持IPC包括传统的管道，System V IPC，即消息队列/共享内存/信号量，以及socket中只有socket支持Client-Server的通信方式。Android系统为开发者提供了丰富进程间通信的功能接口，媒体播放，传感器，无线传输。这些功能都由不同的server来管理。开发都只关心将自己应用程序的client与server的通信建立起来便可以使用这个服务。毫无疑问，如若在底层架设一套协议来实现Client-Server通信，增加了系统的复杂性。在资源有限的手机 上来实现这种复杂的环境，可靠性难以保证。
2. 传输性能。socket主要用于跨网络的进程间通信和本机上进程间的通信，但传输效率低，开销大。消息队列和管道采用存储-转发方式，即数据先从发送方缓存区拷贝到内核开辟的一块缓存区中，然后从内核缓存区拷贝到接收方缓存区，其过程至少有两次拷贝。虽然共享内存无需拷贝，但控制复杂。比较各种IPC方式的数据拷贝次数。共享内存：0次。Binder：1次。Socket/管道/消息队列：2次。
3. 安全性。Android是一个开放式的平台，所以确保应用程序安全是很重要的。Android对每一个安装应用都分配了UID/PID,其中进程的UID是可用来鉴别进程身份。传统的只能由用户在数据包里填写UID/PID，这样不可靠，容易被恶意程序利用。而我们要求由内核来添加可靠的UID。

所以，出于可靠性、传输性、安全性。android建立了一套新的进程间通信方式。

五. Android中进程的级别，以及各自的区别。

1、前台进程

用户当前正在做的事情需要这个进程。如果满足下面的条件之一，一个进程就被认为是前台进程：

1. 这个进程拥有一个正在与用户交互的Activity(这个Activity的onResume()方法被调用)。
2. 这个进程拥有一个绑定到正在与用户交互的activity上的Service。
3. 这个进程拥有一个前台运行的Service（service调用了方法startForeground()）.
4. 这个进程拥有一个正在执行其任何一个生命周期回调方法（onCreate(),onStart(),或onDestroy()）的Service。
5. 这个进程拥有正在执行其onReceive()方法的BroadcastReceiver。

通常，在任何时间点，只有很少的前台进程存在。它们只有在达到无法调合的矛盾时才会被杀 - - 如内存太小而不能继续运行时。通常，到了这时，设备就达到了一个内存分页调度状态，所以需要杀一些前台进程来保证用户界面的反应.

2、可见进程

一个进程不拥有运行于前台的组件，但是依然能影响用户所见。满足下列条件时，进程即为可见：

这个进程拥有一个不在前台但仍可见的Activity(它的onPause()方法被调用)。当一个前台activity启动一个对话框时，就出了这种情况。

3、服务进程

一个可见进程被认为是极其重要的。并且，除非只有杀掉它才可以保证所有前台进程的运行，否则是不能动它的。

这个进程拥有一个绑定到可见activity的Service。

一个进程不在上述两种之内，但它运行着一个被startService()所启动的service。
尽管一个服务进程不直接影响用户所见，但是它们通常做一些用户关心的事情（比如播放音乐或下载数据），所以系统不到前台进程和可见进程活不下去时不会杀它。

4、后台进程

一个进程拥有一个当前不可见的activity(activity的onStop()方法被调用)。

这样的进程们不会直接影响到用户体验，所以系统可以在任意时刻杀了它们从而为前台、可见、以及服务进程们提供存储空间。通常有很多后台进程在运行。它们被保存在一个LRU(最近最少使用)列表中来确保拥有最近刚被看到的activity的进程最后被杀。如果一个activity正确的实现了它的生命周期方法，并保存了它的当前状态，那么杀死它的进程将不会对用户的可视化体验造成影响。因为当用户返回到这个activity时，这个activity会恢复它所有的可见状态。

5、空进程

一个进程不拥有入何active组件。

保留这类进程的唯一理由是高速缓存，这样可以提高下一次一个组件要运行它时的启动速度。系统经常为了平衡在进程高速缓存和底层的内核高速缓存之间的整体系统资源而杀死它们。

六. 线程池的相关知识。

Android中的线程池都是之间或间接通过配置ThreadPoolExecutor来实现不同特性的线程池.Android中最常见的四类具有不同特性的线程池分别为FixThreadPool、CachedThreadPool、SingleThreadPool、ScheduleThreadExecutor.

1).FixThreadPool

只有核心线程,并且数量固定的,也不会被回收,所有线程都活动时,因为队列没有限制大小,新任务会等待执行.

优点:更快的响应外界请求.

2).SingleThreadPool

只有一个核心线程,确保所有的任务都在同一线程中按顺序完成.因此不需要处理线程同步的问题.

3).CachedThreadPool

只有非核心线程,最大线程数非常大,所有线程都活动时,会为新任务创建新线程,否则会利用空闲线程(60s空闲时间,过了就会被回收,所以线程池中有0个线程的可能)处理任务.

优点:任何任务都会被立即执行(任务队列SynchronousQueue相当于一个空集合);比较适合执行大量的耗时较少的任务.

4).ScheduledThreadPool

核心线程数固定,非核心线程(闲着没活干会被立即回收)数没有限制.

优点:执行定时任务以及有固定周期的重复任务

七. 内存泄露，怎样查找，怎么产生的内存泄露。

产生的内存泄露

1. 资源对象没关闭造成的内存泄漏
2. 构造Adapter时，没有使用缓存的convertView
3. Bitmap对象不在使用时调用recycle()释放内存
4. 试着使用关于application的context来替代和activity相关的context
5. 注册没取消造成的内存泄漏
6. 集合中对象没清理造成的内存泄漏

查找内存泄漏

查找内存泄漏可以使用Android Stdio 自带的Android Profiler工具,也可以使用Square产品的LeadCanary.

八. Android优化

性能优化

1. 节制的使用Service 如果应用程序需要使用Service来执行后台任务的话，只有当任务正在执行的时候才应该让Service运行起来。当启动一个Service时，系统会倾向于将这个Service所依赖的进程进行保留，系统可以在LRUcache当中缓存的进程数量也会减少，导致切换程序的时候耗费更多性能。我们可以使用IntentService，当后台任务执行结束后会自动停止，避免了Service的内存泄漏。
2. 当界面不可见时释放内存 当用户打开了另外一个程序，我们的程序界面已经不可见的时候，我们应当将所有和界面相关的资源进行释放。重写Activity的onTrimMemory()方法，然后在这个方法中监听TRIM_MEMORY_UI_HIDDEN这个级别，一旦触发说明用户离开了程序，此时就可以进行资源释放操作了。
3. 当内存紧张时释放内存 onTrimMemory()方法还有很多种其他类型的回调，可以在手机内存降低的时候及时通知我们，我们应该根据回调中传入的级别来去决定如何释放应用程序的资源。
4. 避免在Bitmap上浪费内存 读取一个Bitmap图片的时候，千万不要去加载不需要的分辨率。可以压缩图片等操作。
5. 使用优化过的数据集合 Android提供了一系列优化过后的数据集合工具类，如SparseArray、SparseBooleanArray、LongSparseArray，使用这些API可以让我们的程序更加高效。HashMap

工具类会相对比较低效，因为它需要为每一个键值对都提供一个对象入口，而SparseArray就避免掉了基本数据类型转换成对象数据类型的时间。

布局优化

重用布局文件

标签可以允许在一个布局当中引入另一个布局，那么比如说我们程序的所有界面都有一个公共的部分，这个时候最好的做法就是将这个公共的部分提取到一个独立的布局中，然后每个界面的布局文件当中来引用这个公共的布局。

Tips:如果我们要在标签中覆写layout属性，必须要将layout_width和layout_height这两个属性也进行覆写，否则覆写效果将不会生效。

标签是作为标签的一种辅助扩展来使用的，它的主要作用是为了防止在引用布局文件时引用文件时产生多余的布局嵌套。布局嵌套越多，解析起来就越耗时，性能就越差。因此编写布局文件时应该让嵌套的层数越少越好。

举例：比如在LinearLayout里边使用一个布局。里边又有一个LinearLayout，那么其实就存在了多余的布局嵌套，使用merge可以解决这个问题。

仅在需要时才加载布局

某个布局当中的元素不是一起显示出来的，普通情况下只显示部分常用的元素，而那些不常用的元素只有在用户进行特定操作时才会显示出来。

举例：填信息时不是需要全部填的，有一个添加更多字段的选项，当用户需要添加其他信息的时候，才将另外的元素显示到界面上。用VISIBLE性能表现一般，可以用ViewStub。ViewStub也是View的一种，但是没有大小，没有绘制功能，也不参与布局，资源消耗非常低，可以认为完全不影响性能。

```
<ViewStub
    android:id="@+id/view_stub"
    android:layout="@layout/profile_extra"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    />

public void onMoreClick() {
    ViewStub viewStub = (ViewStub) findViewById(R.id.view_stub);
    if (viewStub != null) {
        View inflatedView = viewStub.inflate();
        editExtra1 = (EditText) inflatedView.findViewById(R.id.edit_extra1);
        editExtra2 = (EditText) inflatedView.findViewById(R.id.edit_extra2);
        editExtra3 = (EditText) inflatedView.findViewById(R.id.edit_extra3);
    }
}
```

tips：ViewStub所加载的布局是不可以使用标签的，因此这有可能导致加载出来出来的布局存在着多余的嵌套结构。

高性能编码优化

都是一些微优化，在性能方面看不出有什么显著的提升的。使用合适的算法和数据结构是优化程序性能的最主要手段。

避免创建不必要的对象 不必要的对象我们应该避免创建：

如果有需要拼接的字符串，那么可以优先考虑使用StringBuffer或者StringBuilder来进行拼接，而不是加号连接符，因为使用加号连接符会创建多余的对象，拼接的字符串越长，加号连接符的性能越低。

当一个方法的返回值是String的时候，通常需要去判断一下这个String的作用是什么，如果明确知道调用方会将返回的String再进行拼接操作的话，可以考虑返回一个StringBuffer对象来代替，因为这样可以将一个对象的引用进行返回，而返回String的话就是创建了一个短生命周期的临时对象。

尽可能地少创建临时对象，越少的对象意味着越少的GC操作。

在没有特殊原因的情况下，尽量使用基本数据类型来代替封装数据类型，int比Integer要更加有效，其它数据类型也是一样。

基本数据类型的数组也要优于对象数据类型的数组。另外两个平行的数组要比一个封装好的对象数组更加高效，举个例子，Foo[]和Bar[]这样的数组，使用起来要比Custom(Foo,Bar)[]这样的一个数组高效的多。

静态优于抽象

如果你并不需要访问一个对系那个中的某些字段，只是想调用它的某些方法来去完成一项通用的功能，那么可以将这个方法设置成静态方法，调用速度提升15%-20%，同时也不用为了调用这个方法去专门创建对象了，也不用担心调用这个方法后是否会改变对象的状态(静态方法无法访问非静态字段)。

对常量使用static final修饰符

```
static int intVal = 42;
static String strVal = "Hello, world!";
```

编译器会为上面的代码生成一个初始方法，称为方法，该方法会在定义类第一次被使用的时候调用。这个方法会将42的值赋值到intVal当中，从字符串常量表中提取一个引用赋值到strVal上。当赋值完成后，我们就可以通过字段搜寻的方式去访问具体的值了。

final进行优化:

```
static final int intVal = 42;
static final String strVal = "Hello, world!";
```

这样，定义类就不需要方法了，因为所有的常量都会在dex文件的初始化器当中进行初始化。当我们调用intVal时可以直接指向42的值，而调用strVal会用一种相对轻量级的字符串常量方式，而不是字段搜寻的方式。

这种优化方式只对基本数据类型以及String类型的常量有效，对于其他数据类型的常量是无效的。

使用增强型for循环语法

```
static class Counter {
    int mCount;
}
Counter[] mArray = ...
public void zero() {
    int sum = 0;
    for (int i = 0; i < mArray.length; ++i) {
        sum += mArray[i].mCount;
    }
}
public void one() {
    int sum = 0;
    Counter[] localArray = mArray;
    int len = localArray.length;
    for (int i = 0; i < len; ++i) {
        sum += localArray[i].mCount;
```

```
    }
}
public void two() {
    int sum = 0;
    for (Counter a : mArray) {
        sum += a.mCount;
    }
}
```

zero()最慢，每次都要计算mArray的长度，one()相对快得多，two()fangfa在没有JIT(Just In Time Compiler)的设备上是运行最快的，而在有JIT的设备上运行效率和one()方法不相上下，需要注意这种写法需要JDK1.5之后才支持。

Tips:ArrayList手写的循环比增强型for循环更快，其他的集合没有这种情况。因此默认情况下使用增强型for循环，而遍历ArrayList使用传统的循环方式。

多使用系统封装好的API

系统提供不了的Api完成不了我们需要的功能才应该自己去写，因为使用系统的Api很多时候比我们自己写的代码要快得多，它们的很多功能都是通过底层的汇编模式执行的。 举个例子，实现数组拷贝的功能，使用循环的方式来对数组中的每一个元素一一进行赋值当然可行，但是直接使用系统中提供的System.arraycopy()方法会让执行效率快9倍以上。

避免在内部调用Getters/Setters方法

面向对象中封装的思想是不要把类内部的字段暴露给外部，而是提供特定的方法来允许外部操作相应类的内部字段。但在Android中，字段搜寻比方法调用效率高得多，我们直接访问某个字段可能要比通过getters方法来去访问这个字段快3到7倍。但是编写代码还是要按照面向对象思维的，我们应该在能优化的地方进行优化，比如避免在内部调用getters/setters方法。

九. 插件化相关技术，热修补技术是怎样实现的，和插件化有什么区别

相同点:

都使用ClassLoader来实现的加载的新的功能类，都可以使用PathClassLoader与DexClassLoader

不同点:

热修复因为是为了修复Bug的，所以要将新的同名类替代同名的Bug类，要抢先加载新的类而不是Bug类，所以多做两件事：在原先的app打包的时候，阻止相关类去打上CLASS_ISPREVERIFIED标志，还有在热修复时动态改变BaseDexClassLoader对象间接引用的dexElements，这样才能抢先代替Bug类，完成系统不加载旧的Bug类.

而插件化只是增肌新的功能类或者是资源文件，所以不涉及抢先加载旧的类这样的使命，就避过了阻止相关类去打上CLASS_ISPREVERIFIED标志和还有在热修复时动态改变BaseDexClassLoader对象间接引用的dexElements.

所以插件化比热修复简单，热修复是在插件化的基础上在进行替旧的Bug类

十. 怎样计算一张图片的大小，加载bitmap过程（怎样保证不产生内存溢出），二级缓存，LRUCache算法。

计算一张图片的大小

图片占用内存的计算公式：图片高度 * 图片宽度 * 一个像素占用的内存大小.所以，计算图片占用内存大小的时候，要考虑图片所在的目录跟设备密度，这两个因素其实影响的是图片的高宽，android会对图片进行拉升跟压缩。

加载bitmap过程（怎样保证不产生内存溢出）

由于Android对图片使用内存有限制，若是加载几兆的大图片便内存溢出。Bitmap会将图片的所有像素（即长x宽）加载到内存中，如果图片分辨率过大，会直接导致内存OOM，只有在BitmapFactory加载图片时使用BitmapFactory.Options对相关参数进行配置来减少加载的像素。

BitmapFactory.Options相关参数详解

1. Options.inPreferredConfig值来降低内存消耗。 比如：默认值ARGB_8888改为RGB_565,节约一半内存。
2. 设置Options.inSampleSize 缩放比例，对大图片进行压缩 。
3. 设置Options.inPurgeable和inInputShareable：让系统能及时回 收内存。 A：inPurgeable：设置为True时，表示系统内存不足时可以被回 收，设置为False时，表示不能被回收。 B：inInputShareable：设置是否深拷贝，与inPurgeable结合使用，inPurgeable为false时，该参数无意义。
4. 使用decodeStream代替其他方法。

decodeResource,setImageResource,setImageBitmap等方法

十一. LRUCache算法是怎样实现的。

内部存在一个LinkedHashMap和maxSize，把最近使用的对象用强引用存储在 LinkedHashMap中，给出来put和get方法，每次put图片时计算缓存中所有图片总大小，跟maxSize进行比较，大于maxSize，就将最久添加的图片移除；反之小于maxSize就添加进来。

之前，我们会使用内存缓存技术实现，也就是软引用或弱引用，在Android 2.3（APILevel 9）开始，垃圾回收器会更倾向于回收持有软引用或弱引用的对象，这让软引用和弱引用变得不再可靠。

算法

一. 算法题

m * n的矩阵，能形成几个正方形（2 * 2能形成1个正方形，2 * 3 2个，3 * 3 6个）

计数的关键是要观察到任意一个倾斜的正方形必然唯一内接于一个非倾斜的正方形，而一个非倾斜的边长为k的非倾斜正方形，一条边上有k-1个内点，每个内点恰好确定一个内接于其中的倾斜正方形，加上非倾斜正方形本身，可知，将边长为k的非倾斜正方形数目乘以k，再按k求和即可得到所有正方形的数目。

设2≤n≤m，k≤n-1，则边长为k的非倾斜有(n-k)(m-k)个，故所有正方形有∑(m-k)(n-k)k个例如m=n=4正方形有3*3*1+2*2*2+1*1*3=20个

下面是面试过程中遇到的题目

大多数题目都可以在上面找到答案.

电话面试题

1. ArrayList 和 Hashmap 简单说一些,区别,底层的数据结构.
2. Handler 消息机制
3. 引起内存泄漏的场景
4. 多线程的使用场景?
5. 常用的线程池有哪几种?
6. 在公司做了什么?团队规模?为什么离职?

面试中实际涉及到的问题

第一轮

1. 知道哪些单例模式,写一个线程安全的单例,并分析为什么是线程安全的?
2. Java中的集合有哪些?解释一下HashMap?底部的数据结构?散列表冲突的处理方法,散列表是一个什么样的数据结构?HashMap是采用什么方法处理冲突的?
3. 解释一下什么是MVP架构,画出图解,一句话解释MVP和MVC的区别?
4. Handle消息机制?在使用Handler的时候要注意哪些东西,是否会引起内存泄漏?画一下Handler机制的图解?
5. 是否做过性能优化?已经采取了哪些措施进行优化?
6. 引起内存泄漏的原因是什么?以及你是怎么解决的?

这些问题应该都是比较基础的问题,每个开发者都应该是非常熟悉并能详细叙述的.这一轮的面试官问的技术都是平时用到的.

第二轮

1. 关于并发理解多少?说几个并发的集合?
2. Handler 消息机制图解?
3. 在项目中做了哪些东西?
4. 画图说明View 事件传递机制?并举一个例子阐述
5. 类加载机制,如何换肤,换肤插件中存在的问题?hotfix是否用过,原理是否了解?
6. 说说项目中用到了哪些设计模式,说了一下策略模式和观察者模式?
7. 会JS么?有Hybid开发经验么?
8. 说一下快排的思想?手写代码
9. 堆有哪些数据结构?

对于这轮米那是明显感觉到压力,知识的纵向了解也比较深,应该是个leader.

第三轮

1. 介绍一下在项目中的角色?
2. 遇到困难是怎么解决的?
3. 如何与人相处,与别人意见相左的时候是怎么解决的,并举生活中的一个例子.
4. 有没有压力特别大的时候?