今日头条面试

自我介绍：面试官您好，我是刘世麟，非常荣幸能参加贵公司的面试，下面我简单介绍一下我的个人情况：我从实习到现在一直在致学教育工作，从事 Android 开发，凭借良好的工作能力和沟通能力，连续两年蝉联「优秀员工」称号，在今年初被公司内聘为技术总监助理，协助技术总监开展部门管理和项目推动工作。在工作之外，我喜欢编写技术博客和在 GitHub 上贡献开源代码，目前在 GitHub 上总共拥有 7k 左右的 Star，数篇技术博客也有数十万阅读。我非常地热爱移动开发，早已久仰贵团队对技术的看重，所以希望今天自己面试有好的表现，未来能有幸与您共事。

> 你有什么要问我的吗?
>
> - 对新入公司的员工有没有什么培训项目?
> - 入职后参与的项目是怎样的?

Java 篇

HashMap 的内部结构？内部原理？和 HashTable 的区别，假如发生了 hash 碰撞，如何设计能让遍历效率高?

HashMap 基于 Map 实现，允许 null 的键值，不保证插入顺序，也不保证序列不随时间而变化。其内部是使用一个默认容量为 16 的数组来存储数据的，而数组中每一个元素却又是一个链表的头结点。所以，更加确切的说，HashMap 内部存储结构是使用哈希表的拉链结构（数组 + 链表），这种存储数据的方法叫做拉链法。而链表中的每个结点都是 `Entry` 类型，而 `Entry` 存储的内容包含 key、value、hash 和 next。

工作原理：主要是通过 hash 的方法，通过 `put` 和 `get` 来存取对象。

- 存取对象时，我们将 key-value 传给 `put()` 时，它通过 `hashCode()` 计算 hash 从而得到桶（bucket）的位置，进一步存储。HashMap 会根据当前桶（bucket）的占用情况来自动调整容量（超过负载因子 Load Factor 则 `resize()` 为原来的 2 倍）；
- 获取对象时，我们将 key 传递给 `get()`，它调用 `hashCode()` 计算 hash 从而得到桶（bucket）的位置，并进一步通过 `equals()` 确认键值对。如果发生碰撞的时候，HashMap 将会通过链表把产生碰撞冲突的元素组织起来。在 Java 8 中，如果一个桶（bucket）中碰撞冲突的元素超过某个限制（默认是 8），则使用红黑树来替换链表，从而提高速度。

> 什么是红黑树?
>
> 红黑树本质上就是一种二叉查找树(二叉查找树的插入、删除、查找最好情况为 O(logn)，但极端的斜树为 O(n))，但它在二叉查找树的基础上额外添加了一个颜色做标记，同时具有一定的规则，这些规则让红黑树保证了一种平衡，插入、删除、查找的最坏时间复杂度都是 O(logn)。

```
class Node<T>{
    public T value;
    public Node<T> parent;
    public boolean isRed;
    public Node<T> left;
    public Node<T> right;
}
```

性质:

- 任何一个结点都有颜色，黑色或者红色；
- 根结点是黑色的；
- 父子结点之间不能出现两个连续的红结点；
- 任何一个结点向下遍历到其子孙的叶子结点，所经历的黑结点个数必须相等；
- 空节点被认为是黑色的；

`HashMap` 和 `HashTable` 虽然都实现了 Map 接口，但 HashTable 的实现是基于 `Dictionary` 抽象类。而且 `HashMap` 中可以把 null 作为键值，所以 `HashMap` 判断是否含有某个键是用 `containsKey()` 而不是 `get()`。 `get()` 方法返回 null 的时候，并不能判断是没有键，也可能是这个键对应的值为 null。但 `HashTable` 是不允许的。还有一个区别就是 `HashMap` 是非同步的，在多线程中需要手动同步，而 `HashTable` 是同步的，可以直接用在多线程中。

但实际上，我们在多线程的时候，更加青睐于使用 `ConcurrentHashMap` 而不是 `HashTable`。因为 `HashTable` 使用 `synchronized` 来做线程安全，全局只有一把锁，直接锁住整个 Hash 表，而 `ConcurrentHashMap` 是一次锁一个桶。在线程竞争激烈的情况下 `HashTable` 效率是非常低下的。但即便如此，我们也不能说 `ConcurrentHashMap` 就可以完全替代 `HashTable`。根本在于 `HashTable` 的迭代器是强一致性的，而 `ConcurrentHashMap` 是弱一致性的。

> ConcurrentHashMap 不允许 key 或者 value 为 null。
>
> 对于「强一致性」和「弱一致性」的理解：比如我们往 ConcurrentHashMap 底层数据结构加入一个元素后，get 可能在某段时间内还看不到这个元素。

讲讲 ConcurrentHashMap。

由于 `HashMap` 是一个线程不安全的容器，主要体现在容量大于`总量*负载因子`发生扩容时会出现环形链表从而导致死循环。

因此需要支持线程安全的并发容器 `ConcurrentHashMap` 。

在 JDK 1.7 中，ConcurrentHashMap 仍然是数组加链表，和 HashMap 不一样的是，ConcurrentHashMap 最外层并不是一个大的数组，而是一个 Segment 的数组，每一个 Segment 包含一个和 HashMap 数据结构差不多的链表数组。

ConcurrentHashMap 采用了分段锁的技术，Segment 继承于 ReentrantLock，所以我们可以很方便的对每一个 Segment 上锁。不会像 HashTable 那样不管是 put 还是 get 操作都需要做同步处理，一个线程占用锁访问一个 Segment 时，根本不会影响到其他的 Segment。

在 ConcurrentHashMap 的 get 方法中，非常高效，因为全程不需要加锁。只需要将 key 通过 hash 之后定位到具体的 Segment，再通过一次 hash 定位到具体的元素上。由于 HashEntry 中的 value 属性是用 volatile 关键字修饰的，保证了内存可见性，所以每次获取到的值都是最新值。

虽然对 HashEntry 的 value 采用了volatile 关键字修饰，但并不能保证并发的原子性，所以 put 操作时仍然需要加锁处理。首先是通过 key 的 hash 定位到具体的 Segment，在 put 之前会进行一次扩容校验。这里比 HashMap 要好的一点是：HashMap 是插入元素之后在看是否需要扩容，有可能扩容之后后续就没有插入就浪费了本次扩容，而 HashMap 的扩容是非常消耗性能的。而 ConcurrentHashMap 不一样，它是先将数据插入之后再检查是否需要扩容，之后再做插入。

而在 JDK 1.8 中，抛弃了原有的 Segment 分段锁，而采用了 CAS + synchronized 来保证并发安全性。并把 1.7 中存放数据的 HashEntry 改为了 Node，但作用还是相同的。其中 value 和 next 均用 volatile 保证可见性。

JVM 虚拟机内存结构，以及它们的作用

JVM 内存结构主要由三大块：堆内存、方法区和栈。

每个线程包含一个栈区，栈中只包含基本数据类型和对象的引用，而且每个栈中的数据都是私有的，其他栈不允许访问。此外，还会存放方法的形式参数和引用对象的地址，在使用完后，栈空间会立即回收，堆空间等待 GC。

堆主要用于存放对象，同时也是垃圾收集器管理的主要区域。每个对象会包含一个与之对应的 class 信息，JVM 只有一个堆区（heap）被所有线程共享，堆区不存放基本数据类型和对象引用，只存放对象本身。

而方法区主要用于存放线程所执行的字节码指令和常量池，会被所有线程共享，方法区包含所有的 class 和 static 变量。

讲讲 JVM 的类加载过程 && 双亲委派模型

JVM 的类加载过程分为加载、验证、准备、解析、初始化 5 个阶段。

- 加载：加载阶段由类加载器进行负责，类加载器根据一个类的全限定名读取该类的二进制字节流到 JVM 内部，然后转换为一个对应的 java.lang.Class 对象实例；一个类由类加载器和类本身一起确定，所以不同类加载器加载同一个类得到的 java.lang.Class 也是不同的。
- 验证：验证阶段负责验证类数据信息是否符合 JVM 规范，是否是一个有效的字节码文件；
- 准备：准备阶段负责为类中的 static 变量分配空间，并初始化（与程序无关、系统初始化）；
- 解析：解析阶段负责将常量池中所有的符号引用转换为直接引用；
- 初始化：初始化阶段负责将所有的 static 域按照程序指定操作对应执行（赋值 static 变量，执行 static 块）。

上述阶段通常都是交叉混合允许，没有严格的先后执行顺序。

双亲委派过程：当一个类加载器收到类加载任务的时候，立即将任务委派给它的父类加载器去执行，直到委派给最顶层的启动类加载器为止。如果父类加载器无法加载委派给它的类时，将类加载任务回退到它的下一级加载器去执行。除了启动类加载器以外，每个类加载器拥有一个父类加载器，用户的自定义类加载器的父类加载器是 AppClassLoader。双亲委派模型可以保证全限名指定的类，只被加载一次。双亲委派模型不具有强制性约束，是 Java 设计者推荐的类加载器实现方式。

采用双亲委派模型的原因：比如黑客定义一个 java.lang.String 类，该 String 类和系统 String 类有一样的功能，只是在某个方法比如 `equels()` 中加入了病毒代码，并且通过自定义类加载器加入 JVM 中，如果没有双亲委派模型，那么 JVM 就可能误以为黑客编写的 String 类是系统 String 类，导致「病毒代码」最终被执行。而有了双亲委派模型，黑客定义的 java.lang.String 类就用于不会被加载进内存，因为最顶端的类加载器会加载系统的 String 类，最终自定义的类加载器无法加载 java.lang.String 类。

> 可以通过重写 `loadClass()` 方法，打破双亲委派模型。

谈谈 Java 的 垃圾回收算法

首先我们肯定得需要确定哪些是活着的对象，哪些是可以回收的。

- 引用计数算法：它是判断对象是否存活的基本算法：给每个对象添加一个引用计数器，每当一个地方引用它的时候，计数器就加 1；当引用失效后，计数器值就减 1。但是这种方法有一个致命的缺陷：**当两个对象相互引用时会导致这两个对象都无法被回收。**
- 根搜索算法：但目前主流商用语言都采用根搜索算法来判断对象是否存活。对于程序来说，根对象总是可以被访问的，从这些根对象开始，任何可以被触及的对象都被认为是「活着的」对象，无法触及的对象被认为是垃圾，需要被回收。

对于垃圾回收算法，主要包含标记-清除算法、复制回收算法、标记-整理算法和分代回收算法。

- 标记-清除算法：首先使用根搜索算法标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象。但有两个缺点：
  - 效率问题：标记和清除的效率都不高；
  - 空间问题：标记清除后会产生大量不连续的内存碎片；
- 复制回收算法：将可用内存分为大小相等的两份，在同一时刻只能使用其中的一份。当其中一份内存使用完了，就把还存活的对象复制到另一份内存上，然后将这一份上的内存情况。复制回收算法能有效地避免内存碎片，**但是算法需要把内存一分为二，导致内存使用率大大降低。**
- 标记-整理算法：复制算法在对象存活率较高的情况下会复制很多的对象，效率会很低。标记-整理算法就解决了这样的问题，**同样采用的是根搜索算法进行存活对象标记，但后续是将所有存活的对象都移动到内存的一端，然后清理掉端外界的对象。**
- 分代回收算法：在 JVM 中不同的对象拥有不同的生命周期，因此对于不同生命周期的对象也可以采用不同的垃圾回收方法，以提高效率，这就是分代回收算法的核心思想。

  在不进行对象存活时间区分的情况下，每次垃圾回收都是对整个堆空间进行回收，花费的时间相对会长。同时，因为每次回收都需要遍历所有存活对象，但实际上，对于生命周期长的对象而言，这种遍历是没有效果的，因为可能进行了很多次遍历，但是他们依旧存在。因此，分代垃圾回收采用分治的思想，进行代的划分，把不同生命周期的对象放在不同代上，不同代上采用最适合它的垃圾回收方式进行回收。

  JVM 中共分为三个代：新生代、老年代和持久代。其中持久代主要存放的是 Java 类的类信息，与垃圾收集要收集的 Java 对象关系不大。

  - 新生代：所有新生成的对象首先都是放在新生代的，新生代采用复制回收算法。新生代的目标就是尽可能快速地收集掉那些生命周期短的对象。新生代按照 8:1 的比例分为一个 Eden 区和两个 Survivor 区。**大部分对象在 Eden 区生成，当 Eden 区满时，还存活的对象将被复制到其中的一个 Survivor 区，当这个 Survivor 区满时，此区的存活对象将被复制到另外一个 Survivor 区，当另一个 Survivor 区也满了的时候，从第一个 Survivor 区复制过来的并且此时还存活的对象，将被复制到了「年老区」**。需要注意，Survivor 的两个区是对称的，没有任何的先后关系，所以同一个区中可能同时存在 Eden 复制过来的对象，和从前一个 Survivor 区复制过来的对象，而复制到年老区的只有从第一个 Survivor 区过来的对象，而且，Survivor 区总有一个是空的。
  - 老年代：在新生代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到老年代中，老年代采用标记整理回收算法。因此，可以认为老年代中存放的都是一些生命周期较长的对象。
  - 持久代：用于存放静态文件，如 final 常量、static 常量、常量池等。持久代对垃圾回收没

有显著影响，但有些应用可能动态生成或者调用一些 class。在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。

谈谈 Java 垃圾回收的触发条件

Java 垃圾回收包含两种类型：Scavenge GC 和 Full GC。

- Scavenge Gc：一般情况下，当新对象生成，并且在 Eden 申请空间失败的时候，就会触发 Scavenge GC，对 Eden 区进行 GC，清除非存活的对象，并且把尚且存活的对象移动到 Survivor 区，然后整理 Survivor 的两个区。这种方式的 GC 是对新生代的 Eden 区进行，不会影响到老年代。因为大部分对象都是从 Eden 区开始的，同时 Eden 区不会分配的很大，所以 Eden 区的 GC 会频繁进行。
- Full GC：Full GC 将会对整个堆进行整理，包括新生代、老年代和持久代。Full GC 因为需要对整个堆进行回收，所以比 Scavenge GC 要慢，因此应该尽量减少 Full GC 的次数。在对 JVM 调优的过程中，很大一部分工作就是对 Full GC 的调节，有如下原因可能导致 Full GC:
  - 老年代被写满；
  - 持久代被写满；
  - System.gc() 被显示调用；

synchronized 和 Lock 的区别

1. 使用方法的区别：
   - synchronized：在需要同步的对象中加入此控制，`synchronized` 可与加在方法上，也可以加在特定代码块中，括号中表示需要加锁的对象。
   - Lock：需要显示指定起始位置和终止位置。一般使用 `ReentrantLock` 类作为锁，多个线程中必须要使用一个 `ReentrantLock` 类作为对象才能保证锁的生效。且在加锁和解锁处需要通过 `lock()` 和 `unlock()` 显式指出。所以一般会在 `finally` 块中写 `unlock()` 以防死锁。
2. 性能的区别：

   `synchronized` 是托管给 JVM 执行的，而 `lock` 是 Java 写的控制锁的代码。在 Java 1.5 中，`synchronized` 是性能低下的。因为这是一个重量级操作，需要调用操作接口，导致有可能加锁消耗的系统时间比锁以外的操作还多。相比下使用 Java 提供的 Lock 对象，性能更低一些。但是到了 Java 1.6，发生了变化。`synchronized` 在语义上很清晰，可以进行很多优化，有适应自旋、锁消除、锁粗化、轻量级锁、偏向锁等，导致在 Java 1.6 上 `synchronized` 的性能并不比 Lock 差。
   - synchronized：采用的是 CPU 悲观锁机制，即线程获得的是独占锁。独占锁就意味着 **其他线程只能依靠阻塞来等待线程释放所。**而在 CPU 转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起 CPU 频繁的上下文切换导致效率很低。
   - Lock：采用的是乐观锁的方式。所谓乐观锁就是：**每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。**乐观锁实现的机制就是 CAS 操作。我们可以进一步研究 `ReentrantLock` 的源代码，会发现其中比较重要的获得锁的一个方法是 `compareAndSetState`。这里其实就是调用的 CPU 提供的特殊指令。
3. `ReentrantLock`：具有更好的可伸缩性：比如时间锁等候、可中断锁等候、无块结构锁、多个条件变量或者锁投票。

volatile 的作用，为什么会出现变量读取不一致的情况，与 synchronized 的区别；

- volatile 修饰的变量具有可见性

  volatile 是变量修饰符，它修饰的变量具有可见性，Java 中为了加快程序的运行效率，对一些变量的操作通常是在该线程的寄存器或是 CPU 缓存上进行的，之后才会同步到主存中，而加了 volatile 修饰符的变量则是直接读取主存，保证了读取到的数据一定是最新的。

- volatile 禁止指令重排

  指令重排是指处理器为了提高程序效率，可能对输入代码进行优化，它不保证各个语句的执行顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。

- 而 synchronized 可用作于一段代码或方法，既可以保证可见性，又能够保证原子性。

在性能方面，synchronized 关键字是防止多个线程同时执行一段代码，会影响程序执行效率，而 volatile 关键字在某些情况下性能要优于 synchronized。

> 可见性是指多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看到修改的值。
>
> 指令重排序：一般来说，处理器为了提供程序运行效率，可以会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。它不会影响到单线程的执行，却会影响到多线程的并发执行。

++i 在多线程环境下是否存在问题，怎么解决?

虽然递增操作 ++i 是一种紧凑的语法，使其看上去只是一个操作，但这个操作并非源自的。因为它并不能作为一个不可分割的操作来执行。实际上，它包含了 3 个独立的操作：读取 i 的值，将值加 1，然后将计算结果返回给 i。这是一个「读取-修改-写入」的操作序列，并且其结果状态依赖于之前的状态，所以在多线程环境下存在问题。

要解决自增操作在多线程下线程不安全的问题，可以选择使用 Java 提供的原子类，如 AtomicInteger 或者使用 synchronized 同步方法。

> 原子性：在 java 中，对基本数据类型的变量的读取和赋值操作是原子性操作，即这些操作是不可被中断的，要么执行，要么不执行。**也就是说，只有简单的读取、赋值（而且必须是将数字赋值给某个变量）才是原子操作。（变量之间的相互赋值不是原子操作，比如 y = x，实际上是先读取 x 的值，再把读取到的值赋值给 y 写入工作内存）**

Thread.sleep() 和 Thread.yield() 区别

`sleep()` 和 `yield()` 都会释放 CPU。

`sleep()` 使当前线程进入停滞状态，所以执行 `sleep()` 的线程在指定的时间内肯定不会执行；`yield()` 只是使当前线程重新回到可执行状态，所以执行 `yield()` 的线程有可能在进入到可执行状态后马上又被执行。

`sleep()` 可使优先级低的线程得到执行的机会，当然也可以让同优先级和高优先级的线程有执行的机会；`yield()` 只能使同优先级的线程有执行的机会。

讲讲常用的容器类

- List 接口实现：允许数据重复
    1. ArrayList：实现 List 接口，内部由数组实现，可以插入 null，元素可重复，线程不安全，访问元素快；空间不足的时候增长率为当前长度的 50%。
    2. Vector：ArrayList 的线程安全写法，由于支持多线程，所以性能比 ArrayList 较低；空间不足的时候增长率为当前长度的 100%。它的同步是通过 Iterator 方法加 synchronized 实现的。
    3. Stack：线程同步，继承自 Vector，添加了几个方法来完成栈的功能。
    4. LinkList：实现 List 接口，内部实现是双向链表，擅长插入删除，元素可重复；线程不同步。
- Set 接口实现：不允许数据重复，最多允许一个 null 元素。
    1. HashSet：实现 Set 接口，线程不同步，不允许重复元素，基于 HashMap 存储，遍历时不保证顺序，并且不保证下次遍历顺序和之前一样，允许 null 元素；
    2. LinkedHashSet：继承自 HashSet，不允许重复元素，可以保持顺序的 Set 集合，基于 LinkedHashMap 实现；
    3. TreeSet：线程不同步，不允许重复元素，保持元素大小次序的集合，里面的元素必须实现 Comparable 接口，源码算法基于 TreeMap;
    4. EnumSet：线程不同步，内部使用 Enum 数组实现，速度比 HashSet 快。**只能存储在构造函数传入的枚举类的枚举值。**
- Map
    1. HashMap：线程不冲突。根据 key 的 hashcode 进行存储，内部使用静态内部类 Node 的数组进行存储，默认初始大小为 16，每次扩大一倍。当发生 hash 冲突时，采用拉链法存储。**可以接受 null 的 key 和 value。**在 JDK 1.8 中，当单个桶中元素大于等于 8 时，链表实现改为红黑树实现；当元素个数小于 6 时，变回链表实现，由此来防止 hashCode 攻击。
    2. LinkedHashMap：继承自 HashMap，相对 HashMap 来说，遍历的时候具有顺序，可以保证插入的顺序，存储方式和 HashMap 一样，采用哈表表方法存储，不过 LinkedHashMap 多维护了一份上下指针；大多数情况下遍历速度比 HashMap 慢，不过有种情况例外，当 HashMap 容量很大，实际数据较少的时候，遍历起来可能会比 LinkedHashMap 慢，因为 LinkedHashMap 的遍历速度只和实际数据有关，和容量无关，而 HashMap 的遍历速度和它的容量有关。
    3. TreeMap：线程不同步，基于红黑树的 NavigableMap 实现，**能够把它保存的记录根据键排序，默认是按键值的升序排序，也可以指定排序的比较器，当用 Iterator 遍历 TreeMap 时，得到的记录是排过序的。**
    4. HashTable：线程安全，不能存储 null 的 key 和 value。

如何去除 ArrayList 的重复元素?

直接采用 HashSet 即可。作为它的参数，然后再 addAll。但这种方式不能保证原来的顺序，如果要求顺序，可以使用 LinkedHashSet 即可。

讲讲 Java 的泛型擦除，泛型主要是为了解决什么问题？如何用泛型做 Json 的解析的?

泛型信息只存在于代码编译阶段，在进入 JVM 之前，与泛型相关的信息会被擦除掉，这样的现象就叫泛型擦除。

泛型的最大作用是提升代码的安全性和可读性。如果没有泛型，我们只能采用 Object 来实现参数的任意化，这样在使用的时候需要进行类型强转，而且这样即使错误了，也不会在编译时就提示错误。而有了泛型就可以很好的解决这个问题，在编译时就会出现编译不通过的现象。而泛型提升代码效率的优点也显而易见，如果没有泛型，解决前面的问题只能通过编写多个 set 和 get 方法来处理，但有了泛型便只需要一个 get 和一个 set 方法就可以处理。

既然说到了「泛型擦除」的概念，就不得不提到一个经典的现象，我们假设两个 ArrayList，一个里面存放 Integer 类型，一个里面存放 String 类型，但调用他们的 `getClass()` 方法却是相等的。因为在 JVM 中它们的 Class 都是 List.class。而我们在做 Json 解析的时候，一定会遇到有的数据是 JsonObject，有个数据是 JsonArray 的情况。我们正常使用 gson 做解析的时候，需要对象.class 做参数传入，所以我们必须要知道 List 里面存放的数据类型。这时候我们就可以通过「反射」的机制来获取里面的泛型信息了。主要方式是采用 `((ParameterizedType)getClass().getGenericSuperclass()).getActualTypeArguments()` 拿到装载的泛型类的真实类型数组，拿到真实类型后我们便可以采用 gson 进行 Json 解析了。

谈谈 Java 的 Error 和 Exception 的区别联系。

Error 和 Exception 均集成自 Throwable，但 Error 一般指的是和虚拟机相关的问题，比如系统崩溃，虚拟机错误，OOM 等，遇到这样的错误，程序应该被终止。而 Exception 表示程序可以处理的异常，可以捕获并且可能恢复。

软引用和弱引用的区别?

只具有弱引用的对象拥有更短暂的生命周期，可能随时被回收(主要发生在每次 GC)。而只具有软引用的对象只有在内存吃紧的时候才会被回收（主要在 OOM 之前），在内存足够的时候，通常不被回收。

比如我们经常去读取一些图片，由于读取文件需要硬件操作，速度较慢，从而导致性能较低。所以我们可以考虑把这些文件缓存起来，需要的时候直接从内存读取。但由于图片占用内存空间较大，比较容易发生 OOM，所以我们可以考虑软引用来避免这个问题发生。

成员变量和静态方法可以被重写么？重写的规则是怎样的?

子类重写父类的方法，**只有实例方法可以被重写**，重写后的方法必须仍为实例方法。**成员变量和静态方法都不能被重写，只能被隐藏。**（形式上可以写，但本质上并不是重写，而是属于隐藏）

方法的重写（override）遵循两同两小一大原则：

- 方法名必须相同，参数类型必须相同。
- 子类返回的类型必须小于或者等于父类方法返回的类型。
- 子类抛出的异常必须小于或者等于父类方法抛出的异常。
- 子类访问的权限必须大于或者等于父类方法的访问权限。

重写方法可以改变其他的方法修饰符，比如 final，synchronized，native 等。

内部类访问局部变量的时候，为什么变量必须加上 final 修饰符?

因为生命周期不同。局部变量在方法结束后就会被销毁，但内部类对象并不一定，这样就会导致内部类引用了一个不存在的变量。所以编译器会在内部类中生成一个局部变量的拷贝，这个拷贝变量的生命周期和内部类对象相同，就不会出现上述问题。但这样就导致了其中一个变量被修改，两个变量值可能不同的问题。为了解决这个问题，编译器就要求局部变量需要被 final 修饰，以保证两个变量值

相同。

Android 篇

Android 为什么推荐使用 ArrayMap 和 SparseArray?

在 Java 中，我们通常会采用 HashMap 来存储 K-V 数据类型，然后在 Android Studio 中这样使用却经常会得到一个警告，提示使用 ArrayMap 或者 SparseArray 做替代。

HashMap 基本上就是一个 HashMap.Entry 的数组，更准确地说，Entry 类中包含以下字段：

- 一个非基本数据类型的 key
- 一个非基本数据类型的 value
- 保存对象的哈希值
- 指向下一个 Entry 的指针

当有键值对插入时，HashMap 会发生什么？

- 首先，键的哈希值被计算出来，然后这个值会赋给 Entry 类中对应的 hashCode 变量。
- 然后，使用这个哈希值找到它将要被存入的数组中“桶”的索引。
- 如果该位置的 “桶” 中已经有一个元素，那么新的元素会被插入到 “桶” 的头部，next 指向上一个元素——本质上使“桶”形成链表。

现在，当你用 key 去查询值时，时间复杂度是 O(1)。

虽然时间上 HashMap 更快，但同时它也花费了更多的内存空间。

所以它会有严重的缺点：

- 自动装箱的存在意味着每一次插入都会有额外的对象创建，这跟垃圾回收机制一样也会影响到内存的利用；
- HashMap.Entry 对象本身是一层额外需要被创建以及被垃圾回收的对象；
- 「桶」在 HashMap 每次被压缩或者扩容的时候都会被重新安排，这个操作会随着对象数量的增长而变得开销极大。

而 ArrayMap 和 SparseArray 更加考虑内存优化，它们内部均采用两个数组进行数据存储。

在 ArrayMap 中，内部的数组一个用于记录 key 的 hash 值，另外一个数组用于记录 value 值，这样既能避免为每个存入 map 中的键创建额外的对象，还能更积极地控制这些数组长度的增加。因为增加长度只需拷贝数组中的键，而不是重新构建一个哈希表。

当插入一个键值对时：

- 键/值被自动装箱。
- 键对象被插入到 mArray[] 数组中的下一个空闲位置。
- 值对象也会被插入到 mArray[] 数组中与键对象相邻的位置。
- 键的哈希值会被计算出来并被插入到 mHashes[] 数组中的下一个空闲位置。

对于查找一个 key :

- 键的哈希值先被计算出来
- 在 mHashes[] 数组中二分查找此哈希值。这表明查找的时间复杂度增加到了 O(logN)。
- 一旦得到了哈希值所对应的索引 index，键值对中的键就存储在 mArray[2*index]，值存储在 mArray[2*index+1]。

- 这里的时间复杂度从 O(1) 上升到 O(logN)，但是内存效率提升了。当我们在 100 左右的数据量范围内尝试时，没有耗时的问题，察觉不到时间上的差异，但我们应用的内存效率获得了提高。

需要注意的是，ArrayMap 并不适用于可能含有大量条目的数据类型。它通常比 HashMap 要慢，因为在查找时需要进行二分查找，增加或删除时，需要在数组中插入或删除键。对于一个最多含有几百条目的容器来说，它们的性能差异并不巨大，相差不到 50%。

Bitmap 加载发生 OOM 怎么办？怎么处理。

在 Android 开发中，Bitmap 的加载通常采用 `BitmapFactory.decodeXXX()`，但这些方法在构造 Bitmap 的时候就开始分配内存，所以很容易造成 OOM，解决方案也很简单，只需要对 `BitmapFactory.Options` 定义属性后进行压缩即可。需要设置 `inJustDecodeBounds` 属性为 true，来阻止解析时分配内存，此时解析返回 null，但是却可以拿到图片的宽高等信息。这个时候可以根据自己的需求通过计算 `inSampleSize` 的值，之后把 `inJustDecodeBounds` 属性设置为 false 后再解析一次即可。

有了解过 IntentService 么?

`IntentService` 作为 `Service` 的子类，默认给我们开启了一个工作线程执行耗时任务，并且执行完任务后，会自动停止服务。拓展 `IntentService` 也比较简单，提供一个构造方法和实现 `onHandleIntent()` 方法就是了，不需要重写父类的其他方法。但如果要绑定服务的话，还是要重写 `onBind()` 返回一个 `IBinder` 的。**使用 Service 可以同时执行多个请求，而使用 IntentService 只能同时执行一个请求。**

详情可点击：面试：Android Service，你真的了解了么?

Activity 的几种启动模式有了解么？各自的使用场景?

详情可点击：面试：说说 Activity 的启动模式

- standard

  Android 默认的启动模式，每次 start 都会新建实例，适用于大多数场景。

- singleTop

  如果 start 的 Activity 刚好在栈顶，则不会新建实例，直接调用 `onNewIntent()` 。使用场景：资讯类内容页面。

- singleTask

  调用 start 启动的 Activity 只要在当前 Activity 栈里面存在，则直接调用 `onNewIntent()` ，并把上面的所有实例全部移除。使用场景：APP 的主页面。

- singleInstance

  Activity 栈里面只会存在一个实例，每次启动都会直接调用 `onNewIntent()` 。适用于比如接电话，闹钟。

Looper.prepare() 和 Looper.loop() 方法分别做了什么?

- Looper.prepare()：首先从 `ThreadLocal` 中获取一个 `Looper`，如果没有则向 `ThreadLocal` 中添加一个新的 `Looper`，同时新建一个 `MessageQueue`。主线程的 `Looper` 在 `ActivityThread` 中创建。

> `ThreadLoacl` 是 Java 提供的用于保存同一进程中不同线程数据的一种机制。每个线程中都保有一个 `ThreadLocalMap` 的成员变量，`ThreadLoaclMap` 内部采用 `WeakRefrence` 数组保存，数组的 key 即为 `ThreadLocal` 内部的 hash 值。**通常情况下，我们创建的变量是可以被任何一个线程访问并修改的，但 ThreadLocal 创建的变量只能被当前线程访问，其他线程无法访问和修改。Handler 正是利用它的这一特性，来做到每个线程都有自己独有的 Looper。**
>
> `ActivityThread` 是 Android 应用的主线程，在 Application 进程中管理执行主线程，调度和执行活动和广播，和活动管理请求的其他操作。

- Looper.loop()：循环使用 `MessageQueue.next()` 来获取消息，该函数在 `MessageQueue` 中没有消息的时候会阻塞，这里采用了 `epoll` 的 I/O 多路复用机制，当获取到一个消息的时候会返回。

讲讲 LruCache，除此之外，你还知道哪些缓存算法?

LruCache 中维护了一个集合 `LinkedHashMap`，该 `LinkedHashMap` 是以访问顺序排序的。当调用 `put()` 方法时，就会在结合中添加元素，并调用 `trimToSize()` 判断缓存是否已满，如果满了就用 `LinkedHashMap` 的迭代器删除队尾元素，即近期最少访问的元素。当调用 `get()` 方法访问缓存对象时，就会调用 `LinkedHashMap` 的 `get()` 方法获得对应集合元素，同时会更新该元素到队头。

除了我们常用的 LRU 缓存，实际上我们在近来停更的 ImageLoader 库里面可以看到其他的缓存算法，比如根据缓存对象被使用的频率来处理的 LFU 算法；比如根据给对象设置失效期的 Simle time-based 算法；比如超过指定缓存，就移除栈内最大内存的缓存对象的 LargestLimitedMemoryCache 算法。

你是如何进行 APK 瘦身的?

首先我们得知道 APK 文件里面都是由哪些文件构成的，通过 APK Analyzer 我们可以得知，APK 占用最大的两块是 lib 和 res。

- 在满足要求的情况下，我们可以指定更少的 so 库进行优化，通常情况下直接指定 armeabi 或者 armeabi-v7a 就可以了。
- 对于图片，我们可以通过 ImageOptim 进行图片压缩，虽然官方提供了 shrinkResources 设置项，但由于该项设置有风险就没有处理。
- 对于 dex 文件，删除了不少无用库（这些库都是早期为了兼容低版本手机）

简述 Android 事件传递机制，什么时候会触发 ACTION_CANCLE。

详情可点击：面试：从源码的角度谈谈 Android 的事件传递机制

我们先说 `ACTION_DOWN` 事件。当用户按下屏幕的时候，事件产生并传递给了 Activity 并调用 Activity 的 `dispatchTouchEvent()` 方法，如果 Activity 没有调用 `onInterceptTouchEvent()` 进行事件拦截，则会传递给 `DecorView`。由于 `DecorView` 继承自 `FrameLayout`，而 `FrameLayout` 是 `ViewGroup` 的子类，所以直接会调用 `ViewGroup` 的 `dispatchTouchEvent()` 方法。在 `ViewGroup` 中同样要调用 `onInterceptTouchEvent()` 查看是否要拦截，默认返回 false，不过我们可以直接改写覆盖它。如果 `ViewGroup` 没有做事件拦截，则会通过一个 for 循环倒序遍历该 `ViewGroup` 下的所有子 View 的 `dispatchTouchEvent()`，在该方法中，会查看该 View 是否 enable，再查看是否重写了 `OnTouchListener` 的 `onTouch()` 方

法，这也是 `onTouch()` 优先于 `onTouchEvent()` 的原因。在 View 的 `onTouchEvent()` 方法中，只要 View 的 `CLICKABLE` 或者 `LONG_CLICKABLE` 有一个为 true，那么 `onTouchEvent()` 就会消耗这个事件，接着就会在 `ACTION_UP` 事件中调用 `performClick()` 方法。如果子 View 没有消耗这个事件，则会传递回给 `ViewGroup`。如果 `ViewGroup` 也没有消耗这个事件，则向上传递给 Activity。

而对 `ACTION_UP` 和 `ACTION_MOVE` 事件就不一样了。不管 `ACTION_DOWN` 事件在哪个控件消费了（return true），那么 `ACTION_UP` 和 `ACTION_MOVE` 就会从上往下（通过 `dispatchTouchEvent()` 方法）做事件分发向下传，就只会传递到这个控件，而不会选择继续往下传递。如果 `ACTION_DOWN` 事件是在 `dispatchTouchEvent()` 消费，那么事件到此停止传递；如果 `ACTION_DOWN` 事件是在 `onTouchEvent()` 消费，那么就会把 `ACTION_UP` 和 `ACTION_MOVE` 传递给该控件的 `onTouchEvent()` 处理并结束传递。

> 处理滑动冲突：主要分为外部拦截法和内部拦截法。
>
> - 外部拦截法：通过重写父 View 的 `onInterceptTouchEvent()`，根据业务逻辑需要在 ACTION_MOVE 里面进行处理。
> - 内部拦截法：通过重写子 View 的 `dispatchTouchEvent()`，根据业务逻辑决定自己消费还是父 View 处理，主要通过 View 的 `requestDisallowInterceptTouchEvent()` 来处理。

对于 `ACTION_CANCLE` 的调用时机，我之前看过系统文档的解释是这样的：在设计设置页面的滑动开关的时候，如果不监听 `ACTION_CANCEL`，在滑动到中间时，如果你手指上下移动，就是移动到开关控件之外，则此时会触发 `ACTION_CANCLE` 而不是 `ACTION_UP`，造成开关的按钮停顿在中间位置。实际上我也去写了一个小 demo 做尝试，我们知道 `ViewGroup` 是一个能放置其他 View 的布局类，在每次事件分发的过程中，它都会调用自己的 `onInterceptTouchEvent()` 来判断是否拦截事件。假如前面一直返回 false 在让子 View 处理事件，这时候突然 `onInterceptTouchEvent()` 返回 true 进行事件拦截，这时候便会在子 View 中响应 `ACTION_CANCEL` 了。

Android 的多点触摸机制

简述 Android 的绘制流程。

View 的绘制流程是从 `ViewRootImpl` 的 `performTraversals()` 方法开始，其内部会调用 performMeasure()、performLayout()、performDraw()。

- performMeasure()：

  performMeasure() 会调用最外层的 ViewGroup 的 measure()，measure() 会回调 onMeasure()。ViewGroup 的 onMeasure() 是抽象方法，但其提供了 measureChildren()。这会遍历子 View 然后循环调用 measureChild()，measureChild() 中会用 getChildMeasureSpec()、父 View 的 MeasureSpec 和子 View 的 LayoutParam 一起获取本 View 的 MeasureSpec。然后再调用 View 的 measure()，View 的 measure() 再调用 Viwe 的 onMeasure()。该方法默认返回的是 measureSpec 的测量值，所以我们要实现自定义的 wrap_content 需要重写该方法。

  - `MeasureSpec`（View 的内部类）测量规格为 int型，值由高 2 位规格模式 `specMode` 和低 30 位具体尺寸 `specSize` 组成。其中 specMode 只有三种值：

        - `MeasureSpec.EXACTLY` //确定模式，父 View 希望子 View 的大小是确定的，由 specSize 决定；
        - `MeasureSpec.AT_MOST` //最多模式，父 View 希望子 View 的大小最多是 specSize 指定的值；
        - `MeasureSpec.UNSPECIFIED` //未指定模式，父View完全依据子View的设计值来决定；
    - 最顶层 DecorView 测量时的 MeasureSpec 是由 ViewRootImpl 中 getRootMeasureSpec() 方法确定的，**LayoutParams 宽高参数均为MATCH_PARENT，specMode 是 EXACTLY，specSize 为物理屏幕大小。**
    - 只要是 ViewGroup 的子类就必须要求 LayoutParams 继承子 MarginLayoutParams，否则无法使用 layout_margin 参数。
    - View 的布局大小由父 View 和子 View 共同决定。
    - 使用 View 的 `getMeasuredWidth()` 和 `getMeasuredHeight()` 方法来获取 View 测量的宽高，必须保证这两个方法在 `onMeasure()` 流程之后被调用才能返回有效值。

    > - View 的测量宽高和实际的宽高有区别么?
    >
    >   基本上 99% 都是没区别的，但存在特殊情况，有时候可能会因为某种原因对 View 进行多次测量，这样每次测量的大小可能是不相等的，但这样的情况下，最后一次测量基本是一致的。
    > - View 的 MeasureSpec 由谁决定?
    >
    >   除了 DecorView 以外，其它 View 的 MeasureSpec 都是由自己的 LayoutParams 和父容器一起决定的。而 DecorView 的测量存在于 ViewRootImpl 的源码中。
    > - 自定义 View 中如何没有处理 wrap_content 情况，会发生什么？为什么？如何解决?
    >
    >   会发生设置成 match_parent 一样的效果，View 设置为 wrap_content，实际上就是 MeasureSpec 的 AT_MOST 模式，如果没有处理，测量出来的宽高则会是测量的父布局的剩余容量大小，实际上这样是和设置为 match_parent 是一样的。解决方案就是给自定义 View 设置一个默认的大小，这样的话在 wrap_content 的时候就显示默认的大小了。

- performLayout()：

  performLayout() 会调用最外层 ViewGroup 的 layout() 方法，layout() 方法通过调用抽象方法 onLayout() 来确定子 View 的位置。

    - 使用 View 的 getWidth() 和 getHeight() 方法来获取 View 测量的宽高，必须保证这两个方法在 `onLayout()` 流程之后被调用才能返回有效值。
- perfomrDraw()：

  performDraw() 方法会调用最外层的 ViewGroup 的 draw()，其中会先后绘制背景、绘制自己、绘制子 View、绘制装饰。

  ViewRootImpl 中的代码会创建一个 Canvas 对象，然后调用 View 的 draw() 方法来执行具体的绘制工作。主要点为：

    - 如果该 View 是一个 ViewGroup，则需要递归绘制其所包含的所有子 View。
    - View 默认不会绘制任何内容，真正的绘制都需要自己在子类中实现。
    - View 的绘制是借助 `onDraw()` 方法传入的 `Canvas` 类来进行的。

- 在获取画布剪切区（每个 `View` 的 `draw` 中传入的 `Canvas` ）时会自动处理掉 `padding`，子 `View` 获取 `Canvas` 不用关注这些逻辑，只用关心如何绘制即可。
- 默认情况下子 `View` 的 `ViewGroup.drawChild` 绘制顺序和子 `View` 被添加的顺序一致，但是你也可以重载 `ViewGroup.getChildDrawingOrder()` 方法提供不同顺序。

Android 的进程间通信，Linux 操作系统的进程间通信。

Linux 操作系统的所有进程间通信 IPC 包括：

1. 管道：在创建时分配一个 page 大小的内存，缓存区大小比较有限；
2. 消息队列：信息复制两次，额外的 CPU 小号，不适合频繁或者信息量大的通信；
3. 共享内存：无需复制，共享缓冲区直接附加到进程虚拟地址空间，速度快。但进程间的同步问题操作系统无法实现，必须各进程利用同步工具解决；
4. 套接字：作为更通用的接口，传输效率低，主要用于不同机器或跨网络的通信；
5. 信号量：常作为一种锁机制，防止某进程正在访问共享资源时，其它进程也访问该资源。因此它主要作为进程间以及同一进程内不同线程之间的同步手段；
6. 信号：不适用于信息交换，更适用于进程中断控制，比如非法内存访问，杀死某个进程等。

Android 内核也是基于虚拟机内核，但它的 IPC 通信却采用 Binder，是有原因的：

- 性能优越：Binder 数据拷贝仅需一次，而管道、消息队列、Socket 均需要两次，所以 Binder 性能仅次于共享内存（共享内存不需要拷贝数据）。

  > 因为 Linux 内核没有直接从一个用户空间到另一个用户空间直接拷贝的函数，需要先用 `copy_from_user()` 拷贝到内核空间，再用 `copy_to_user()` 拷贝到另外一个用户空间。而在 Android 中，**为了实现用户空间到用户空间的拷贝，`mmap()` 分配的内存除了映射进了接收方进程里，还映射到了内核空间。所以调用 `copy_from_user()` 将数据拷贝进内核空间也相当于拷贝进了接收方的用户空间。**

- 稳定性高：Binder 基于 C/S 架构，稳定性明显优于不分客户端和服务器端的共享内存方式；
- 安全性高：为每个 APP 分配 UID，进程的 UID 是鉴别进程身份的重要标志。

Android 的 IPC 通信方式。

1. 使用 Bundle 的方式

   因为 Bundle 实现了 Parceable 接口，所以可以很方便的在不同进程之间传输，传输的数据必须是能够被反序列化的数据或基本数据类型。**Bundle 的方式简单轻便，但只能单方面传输数据，使用有局限。**所以仅仅适合四大组件的进程间通信。

2. 使用文件共享的方式

   共享文件也是一种不错的进程间通信方式，两个进程通过读写同一个文件来实现交换数据。**但这样的方式在并发编程的时候容易出问题。**所以只适合没有并发的场景交换一些简单的数据。

3. 使用 Messenger 的方式

   Messenger 底层实现是 AIDL，功能一般，支持一对多的**串行通信**，支持实时通信。但不能很好的处理并发现象，不支持 RPC，只能传输 Bundle 支持的数据类型。适用于低并发的一对多即时通信，无 RPC 需求。

4. 使用 AIDL 的方式

AIDL 是一种 IDL 语言，功能强大，支持一对多的**并发通信**，支持实时通信。但使用稍复杂，需要处理好线程同步，适合于一对多通信而且有 RPC 需求。

在我印象中，AIDL 分为两类，一类是定义 Parcelable 对象，以供其他 AIDL 文件使用。另一类是定义方法接口，以供系统使用来完成跨进程通信。

它支持基本数据类型和实现 Parcelable 的类型。传参除了 Java 基本类型和 String、CharSequence 之外其他的都必须在前面加上 tag 参数。分别是：

- in：表示数据只能由客户端流向服务端；
- out：表示数据只能由服务端流向客户端；
- inout：表示数据可以在服务端和客户端双向流通。

5. 使用 ContentProvider 的方式

   ContentProvider 数据访问功能方面特别强大，支持一对多并发数据共享，但却是一种受约束的 AIDL，主要提供数据源的增删查改操作，适用于一对多的进程间数据共享。

6. Socket

   Socket 方式功能强大，可通过网络传输字节流，支持一对多的并发实时通信，但其使用繁琐，不支持直接的 RPC，适用于网络数据交换。

一个 APP 的程序入口是什么？整个 APP 的主线程的消息循环是在哪里创建的?

APP 的程序入口是 ActivityThread.main()。

在 ActivityThread 初始化的时候，就已经创建消息循环了，所以在主线程里面创建 Handler 不需要指定 Looper，而如果在其他线程使用 Handler，则需要单独使用 Looper.prepare() 和 Looper.loop() 创建消息循环。

简述 APP 的启动流程

1. 首先是调用 startActivity(intent)，通过 Binder IPC 机制，最终调用到 ActivityManagerService，这个 Service 首先会通过 PackageManager 的 resolveIntent() 方法来收集这个 intent 对象的指向信息；然后通过 grantUriPermissionLocked() 方法来验证用户是否有足够的权限去调用该 intent 对象指向的 Activity。如果有权限，则开始创建进程。
2. 创建进程。

   ActivityManagerService 调用 startProcessLocked() 方法来创建新的进程，这个方法会通过 socket 通道传递参数给 Zygote 进程，Zygote 孵化自身，并调用 ZygoteInit.main() 方法来实例化 ActivityThread 对象并最终返回新进程的 PID。

   ActivityThread 随后会在 main 方法中依次调用 Looper.prepareLoop() 和 Lopper.loop() 来开启消息循环。这也是在主线程中使用 Handler 并不需要使用这两个方法来开启消息循环的原因。

3. 绑定 Application

   接下来要做的就是将进程和指定的 Application 绑定起来，这个是通过 ActivityThread 对象中调用 bindApplication() 方法完成的，这个方法发送一个 BIND_APPLICATION 的消息到消息队列中，最终通过 handleBindApplication() 方法来处理这个消息，然后调用 makeApplication() 方法来加载 APP 的 classes 到内存中。

4. 启动 Activity

经过前面的步骤，系统已经拥有了该 Application 的进程，后面的话就是普通的从一个已经存在的进程中启动一个新进程的 activity 了。实际调用方法详见 Activity 的启动过程。

简述 Activity 的启动过程。

首先还是得当前系统中有没有拥有这个 Application 的进程。如果没有，则需要处理 APP 的启动过程。在经过创建进程、绑定 Application 步骤后，才真正开始启动 Activity 的方法。 startActivity() 方法最终还是调用的 startActivityForResult()。

在 startActivityForResult() 中，真正去打开 Activity 的实现是在 Instrumentation 的 execStartActivivity() 方法中。

在 execStartActivity() 中采用 checkStartActivityResult() 检查在 manifest 中是否已经注册，如果没有注册则抛出异常。否则把打开 Activity 的任务交给 ActivityThread 的内部类 ApplicationThread，该类实现了 IApplicationThread 接口。这个类完全搞定了 onCreate()、onStart() 等 Activity 的生命周期回调方法。

在 ApplicationThread 类中，有一个方法叫 scheduleLaunchActivity()，它可以构造一个 Activity 记录，然后发送一个消息给事先定义好的 Handler。

这个 Handler 负责根据 LAUNCH_ACTIVITY 的类型来做不同的 Activity 启动方式。其中有一个重要的方法 handleLaunchActivity() 。

在 handleLaunchActivity() 中，会把启动 Activity 交给 performLaunchActivity() 方法。

在 performLaunchActivity() 方法中，首先从 Intent 中解析出目标 Activity 的启动参数，然后用 ClassLoader 将目标 Activity 的类通过类名加载出来并用 newInstance() 来实例化一个对象。

创建完毕后， 开始调用 Activity 的 onCreate() 方法，至此，Activity 被成功启动。

Bundle 的数据结构，如何存储？既然有了 Intent.putExtra()，为何还需要 Bundle?

`Bundle` 内部是采用 `ArrayMap` 进行存储的，并不是 `HashMap`。而 `ArrayMap` 内部是使用两个数组进行数据存储，一个数组记录 key 的 hash 值，另一个数组记录 value 值，内部使用二分法对 key 进行排序，并使用二分法进行添加、删除、查找数据，因此它只适合于小数据量操作。

`Intent` 可以附加的数据类型，大多数都是基础类型，而 `Bundle` 添加数据的功能更加强大。比如有一些可以使用 `Bundle` 的场景：

- Activity A 传递给 Activity B 再传递给 Activity C，如果用 `Intent` 的话，需要不断地取和存，比较麻烦，但用 `Bundle` 就很好解决了这个问题；
- 在设备旋转的时候保存数据，我们就会用 `Bundle`；
- 在 Fragment 中，传递数据采用 `Bundle`。

> 之所以不用 HashMap 是因为 HashMap 内部使用是数组 + 链表的结构（JDK 1.8 增加了红黑树），在数据量较少的情况下，HashMap 的 Entry Array 会比 ArrayMap 占用更多的内存。 而 Bundle 的使用场景是小数据量，实际上 SDK 也对数据量做了限制，在数据量太大的时候使用 Bundle 传递会抛异常。所以相比之下，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。

Android 图片加载框架对比区别。

- ImageLoader：支持下载进度监听，可以在 View 滚动的时候暂停图片的加载，默认实现多种内存缓存的方法，比如最大最先删除、使用最少最先删除、最近最少使用最先删除、先进先删除等，而且也可以自己配置缓存算法，可惜现在已经不再维护，该库使用前还需要进行配置。
- Picasso：包比较小，可以取消不在视野范围内图片资源的加载，并可使用最少的内存完成复杂的图片转换，可以自动添加二级缓存，支持任务调度优先级处理，并发线程数可以根据网络类型进行调整，图片的本地缓存交给了 OkHttp 处理，可以控制图片的过期时间。但功能比较简单，自身并不能实现「本地缓存」的功能。
- Glide：支持多种图片格式缓存，适用于更多的内容表现形式，比如 Gif，WebP、缩略图、Video 等，支持根据 Activity 或者 Fragment 的生命周期管理图片加载请求；对 Bitmap 的复用和主动回收做的较好，缓存策略做的比较高效和灵活（Picasso 只会缓存原始尺寸的图片，而 Glide 缓存采用的是多种规格）。加载速度快且内存开销小（默认 Bitmap 格式的不同，使得内存开销是 Picasso 的一半）。但方法较多较复杂，因为相当于 Picasso 的改进，包较大但总的来说影响不大。

  > Glide 还可以设置 Gif 图次数以及可以设置 Gif 的第一帧显示，比如常见的地址展示 Gif 动画。

- Fresco：最大的优势莫过于在 5.0 以下设备的 Bitmap 加载。在 5.0 以下的系统， Fresco 会将图片放到一个特别的内存区域，大大减少 OOM（会在更底层的 Native 层对 OOM 进行处理，图片将不再占用 APP 的内存），所以它相当适用于高性能加载大量图片的场景。但它的包太大也一直为人诟病，而且用法比较复杂，底层还会涉及 C++ 领域。

讲讲 Glide 的三级缓存

Glide 的三级缓存为内存缓存、磁盘缓存和网络缓存，默认采用的是内存缓存。

有缓存，必定就有缓存 key，之前简单看了一下 key 的生成方法，真的繁琐，传了整整 10 个参数。不过逻辑也不是很复杂，主要就是重写 equals() 和 hashCode() 方法来保证 Key 的唯一性。

- 内存缓存

  Glide 默认会采用内存缓存，当然可以通过 skipMemoryCache(true) 方法禁用。Glide 的内存缓存其实也是使用了 LruCache 算法，它的主要原理就是把最近使用过的对象放在 LinkedHashMap 中，并且把最近最少使用的对象在缓存值达到预设值之前从内存中删除。不过在 Glide 中，除了 LruCache 算法以外，Glide 还结合了一种弱引用机制，共同完成缓存功能。所以在获取的时候，会先用 LruCache 的 loadFromCache() 方法来获取，如果获取到则会将它从缓存区移除，然后再把这个图片存储到 activeResource 中。这个 activeResource 就是一个缓存的 HashMap，用来缓存正在使用中的图片，这样可以保护这些图片不会被 LruCache 算法回收掉。如果没有获取到再通过弱引用机制的 loadFromActiveResources() 方法来获取缓存图片。若是都没有获取到，才会向下执行。

  在 Glide 中，有一个变量，当这个变量大于 0，则代表图片正在使用中，就放到 activeResource 弱引用缓存中。如果这个变量变成 0 以后，我们就从弱引用中移除，并 put 到 LruResourceCache 中。

- 硬盘缓存

  硬盘缓存，有 4 种模式，默认只会缓存转换过后的图片，另外也可以选择只缓存原始图片或者既缓存原始图片也缓存转换过的图片，甚至是选择不缓存。

  和内存缓存类似，硬盘缓存的实现也是采用的 LruCache 算法。Glide 会优先从磁盘缓存中读取图片，只有从缓存中读取不到图片时，才会去读取原始图片。

在缓存原始图片的时候，其实传入的 Key 和之前的 Key 不一样，这是因为原始图片并不需要那么多的参数，所以这个 Key 的生成不需要那么多的参数。

主线程中 Looper.loop() 无限循环为什么不会造成 ANR?

`Looper.loop()` 主要功能是不断地接收事件和处理事件，只能说是某一个消息或者说消息的处理阻塞了 `Looper.loop()`，而不是 `Looper.loop()` 阻塞了它。总的来说，`Looper.loop()` 方法可能会引起主线程的阻塞，但只要它的消息循环没有被阻塞，能一直处理事件就不会造成 ANR。

RxJava 2 的背压是什么？如何形成的？又如何解决?

被观察者发送消息太快以至于它的操作符或者订阅者不能及时处理相关的消息，从而操作消息造成阻塞的现象叫做背压。

在 RxJava 1.0 中，背压事件的缓存尺很小，只有 16，所以不能处理较大量的并发事件，而且被观察者无法得知观察者对事件的处理能力和事件处理进度，只知道把时间一股脑抛给观察者，所以操作很多事件不能被背压，抛出我们闻名的 MissingBackpressureException 异常。

常规解决方案：

- 既然背压是由于被观察者发送事件太快或者太大所导致，所以我们肯定可以通过一定的方法让被观察者发送事件更慢；
- 使用 `filter` 过滤操作符或者 `sample` 操作符让观察者少处理一些事件；

很明显，我们在正常开发中这样的解决方案肯定不可取，因为我们不知道应该让被观察者保持一个怎样的频率才是最佳的，所以在 RxJava 2.0 中官方推出了 `Flowable` 和 `Subscriber` 来支持背压，同时去除了 `Observable` 对背压的支持。`Flowable` 在设计的时候实际上采用了一种新的思路，也就是「响应式拉取」的方式来处理处理事件和发出事件速率不均衡的问题。在 `Flowable` 的使用中，我们会有几种背压策略，并在 `create()` 的时候作为参数传进去。值得注意的是我们可以通过调用 `request()` 方法来定义观察者处理事件的能力，在被观察者上面可以通过该 `requested` 变量来获取下游的处理能力（这个值最大能得到 128，因为它是内部调用的）。这样只要被观察者根据观察者的处理能力来决定发送多少事件，就可以避免发出一堆事件从而导致 OOM。这也就完美的解决了事件丢失的问题，又解决了速度的问题。如果我们没有调用 `request()` 进行设置，依然会出现收不到消息（不在同一个线程，且没有超过缓存区的 128 个）或直接抛出异常（在同一个工作线程）。

RxJava 2.0 背压的策略确实相对 1.0 版本提升了很多，但是真的没有我们想象的完美，因为丢失的事件不一定是我们想要丢失的，所以还是应该根据实际需求来制定防阻塞策略。

> RxJava 2.0 中增加了背压策略模式：
>
> - ERROR：处理跟不上发送速度时报异常；
> - MISSING：如果流的速度无法同步可能会报异常；
> - BUFFER：和 1.0 的 Observable 一样，超过缓存区直接报异常；
> - DROP：跟不上速度直接丢弃；
> - LASTED：一直保留最新值，直到被下游消费掉；

自定义 View 优化

- 频繁调用的方法尽量减少不必要的代码，比如 `onDraw()`，尽量不要做内存分配的事情；
- 尽量地减少 `onDraw()` 的调用次数，如果可以尽量少使用无参数的 `invalidate()`，而选择

4 个参数的 `invalidate()` 进行局部重绘；
- 避免使用 `requestLayout()` ， `requestLayout()` 的执行会去重新 measure ，这样的计算一旦遇到冲突，将需要比较长的时间；

布局优化的措施

1. 降低 Overdraw，减少不必要额背景绘制；
2. 减少嵌套和空间的个数：RelativeLayout 和 LinearLayout 都需要 measure 两次才能完成，所以一定要尽量少嵌套；
    - 使用 merge 标签减少 UI 的层级，提高加载速度，替代 FrameLayout 或者当一个布局 include 另外一个布局的时候，主要用于消除目录结构中的多余 ViewGroup；
    - 使用 ViewStub，作用和 include 类似，也是加载另外一个布局，但和 include 不同的是，ViewStub 引入的布局模式是不显示的，不会占用 CPU 和内存，所以经常用来引入那些默认不显示的布局，比如进度条、错误提示等；

APP 如何保证后台服务不被杀死?

主要是两个方面，提升进程的优先级，降低被杀死的概率，以及在进程被杀死后，进行拉活。

提升进程优先级的方案：

1. 利用 Activity：监控手机锁屏解屏事件，在锁屏时启动 1 个像素的 Activity，解锁的时候把 Activity 销毁，该方案可以使进程的优先级在屏幕锁屏时间由 4 提升为最高优先级 1。
2. 在后台播放一段无声音乐。
3. 利用 Notification：Service 的优先级为 4，可以通过 setForeground 把后台 Service 设置为前台 Service，但是不做处理会是用户感知的。解决方案是通过实现一个内部 Service，在 LiveService 和其内部 Service 中同时发送具有相同 ID 的 Notification，然后将内部 Service 结束掉。随着内部 Service 的结束，Notification 将会消失，但系统优先级依然保持为2。

进程死后拉活的方案：

1. 利用系统广播拉活。缺点是系统广播不可控，只能保证发生事件时拉活，无法保证进程挂掉后立即拉活；而且广播接收器被管理软件、系统软件通过「自启管理」等功能禁用的场景无法接收广播，从而无法重启。所以该方案多用于备用方案。
2. 利用第三方应用广播拉活：比如个推 SDK、友盟等。
3. 利用系统 Service 机制拉活。把 Service 设置为 START_STICKY，利用系统机制在 Service 挂掉后自动拉活，但在短时间只能有 5 次，而且取得 Root 权限的管理工具或者系统工具可以通过 forestop 停止掉，无法重启。

如何定位 ANR，有哪些检测方式

定位：如果在开发机器上出现问题，我们可以通过查看 /data/anr/traces.txt 即可，最新的 ANR 信息会在最开始的部分。或者使用 Systrace 和 TraceView 找出影响响应的问题。

- TraceView 是 Android SDK 自带的一个系统性能分析工具，可以定位应用代码中的耗时操作。
- Systrace 是 Android 4.1 新增的应用性能数据采样和分析工具，需要借助 chrome 浏览器。

检测：可以使用 BlockCanary 分析 Android 的 ANR。

介绍下 MVP 模式，为什么从 MVC 重构到了 MVP？知道 MVVM 么?

MVP 实际上就是 MVC 的变种，MVP 把 Activity/Fragment 中的 UI 逻辑抽象成了 View 接口，把业务逻辑抽象成 Presenter 接口，Model 类还是原来的 Model。主要有以下好处：

- Activity 代码更加简洁，耦合性更低，可读性更高，各板块各司其职；
- 方便进行单元测试，因为Presenter 被抽象成接口，可以有多种具体的实现；
- 可以有效避免内存泄漏。在 MVC 模式中，异步任务对 UI 的操作会放在 Activity 中，所以异步任务会持有 Activity 的引用，这在特定情况下肯定会造成内存泄漏的。

MVVM 虽然没有使用过，但却有所了解，最开始了解还是从 Google 2015 推出的 databinding 开始的。MVVM 包含 View、Model、ViewModel 三个部分。核心思想和 MVP 类似，利用数据绑定、依赖属性、命令、路由事件等新特性，打造了一个更加灵活高效的架构。

- View 对应于 Activity 和 XML，负责 View 的绘制以及用户交互；
- Model 同样是代表实体模型；
- ViewModel：负责 View 和 Model 间的交互，负责业务逻辑。

Parcelable 和 Serializable 的区别。

Parcelable 和 Serializable 都支持序列化和反序列化操作，但 Serializable 使用 I/O 读写存储在硬盘上，而 Parcelable 是直接在内存中读写，Parcelable 的性能比 Serializable 好，在内存开销方面较小，所以在内存间做数据传输时推荐使用 Parcelable，如在 Activity 之间传输数据。

而 Serializable 对数据持久化更方便保存，所以在保存或网络传输数据时选择 Serializable，因为 Android 不同版本 Parcelable 可能不同，所以不推荐使用 Parcelable 进行数据持久化。

System.gc() 和 Runtime.gc() 的区别

`System.gc()` 和 `Runtime.gc()` 是等效的，在 `System.gc()` 内部也是调用的 `Runtime.gc()`。**调用两者都是通知虚拟机要进行 gc，但是否立即回收还是延迟回收，由 JVM 决定。**两者唯一的区别就是一个是类方法，一个是实例方法。

算法 && Other

如何在 100 亿个数中找到最大的 100 个数。

因为数字很多，所以可以先用 hash 法进行去重，如果重复率很高的话，这样可以减少很多的内存用量，从而缩小运算空间。然后再采用最小堆的方式进行处理，即先读取前 100 个数，形成最小堆，保证堆顶的数最小，然后依次读取剩余数字，只要出现比堆顶元素大的元素，则替换堆顶元素并重新调整堆为最小堆，直到读取完成，最小堆中的元素就是最大的 100 个数。

生成最小堆方式：

```
private static void heapAdjustMin(int[] arr, int i, int n) {
    int temp = arr[i];
    // j 代表左结点
    int j = 2 * i + 1;
    while (j < n) {
        // 我们先找出小的，如果满足说明右结点比较小
        if (j + 1 < n && arr[j] > arr[j + 1])
            ++j;
        // 此时 arr[j] 就是最小值了
```

```
            // 如果 父结点比左右结点最小的都小，说明已经符合条件
            if (temp <= arr[j])
                break;
            // 否则的话就把最小的值和 父结点对换；
            arr[i] = arr[j];
            i = j;
            j = 2 * i + 1;
        }
        arr[i] = temp;
    }
```

算法题：一个行列都有序的数组，给定一个数，找出具体的位置

如果都是升序，则从最右边开始找，小了就往下找，大了就往左找。

TCP 和 UDP 区别是什么？简单说一下 TCP 三次握手和四次挥手协议。

TCP 和 UDP 同为数据传输协议，但 TCP 更加强调安全，它面向字节流，需要先通过三次握手协议建立连接后才可以进行数据传输。而 UDP 更强调速度，它面向报文传输(数据报有长度)，无需连接，所以带来的优势是效率高，而安全性低也一直为人诟病。

> TCP 并不能保证数据一定会被对方收到，只能够做到如果有可能，就把数据传递到接收方，否则就通知用户（通过放弃重传并且中断连接这一手段）。因此准确说 TCP 也不是 100% 可靠的协议，它所能提供的是数据的可靠递送或故障的可靠通知。

对于 TCP 三次握手：

- 第一次握手：客户端发送 SYN 报文，并置发送序号为 X，然后客户端进入 SYN_SEND 状态，等待服务器确认；
- 第二次握手：服务器收到 SYN 报文段，然后发送 SYN + ACK 报文，并置发送序号为 Y，在确认序号为 X + 1，发送后服务器进入 SYN_RECV 状态；
- 第三次握手：客户端收到服务器的 SYN + ACK 报文段，然后向服务器发送 ACK 报文，并置发送序号为 Z，在确认序号为 Y + 1，服务器和客户端都进如 ESTABLISHED 状态，完成 TCP 三次握手。

进行三次握手，**主要是为了防止服务器端一直等待而浪费资源。**

对于 TCP 四次挥手：

- 第一次挥手：客户端发送一个 FIN 标志位置为 1 的包，表示自己已经没有数据可以发送，但仍可以接收数据，发送完毕后，客户端进入 FIN_WAIT_1 状态；
- 第二次挥手：服务器确认客户端的 FIN 包，发送一个确认包，表明自己接收到了客户端关闭连接的请求，但还没准备好关闭连接。发送完毕后，服务器端进入 CLOSE_WAIT 状态，客户端接收到这个确认包后，进入 FIN_WAIT_2 状态，等待服务器端关闭连接；
- 第三次挥手：服务器端准备关闭连接时，向客户端发送结束连接请求，FIN 置为 1。发送完毕后，服务器端进入 LAST_ACK 状态，等待来自客户端的最后一个 ACK；
- 第四次挥手：客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入到 TIME_WAIT 状态，等待可能出现的要求重传的 ACK 包；服务器端接收到这个确认包之后，关闭连接，进入 CLOSE 状态；客户端等待了某个固定时间之后，没有收到服务器端的 ACK，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入 CLOSE 状态。

HTTP 状态码

- 1XX：指示信息，表示请求已接收，继续处理
- 2XX：成功：常见 200
- 3XX：重定向：常见 301：请求永久重定向；302：请求临时重定向；
- 4XX：客户端错误：常见 400：客户端请求语法错误，服务器无法理解；401：请求未经授权；403：服务器收到请求，但拒绝服务；
- 5XX：服务器错误：常见 500：服务器发生不可预期的 URL；503：服务器当前不能处理客户端请求，一段时间后恢复正常。

讲讲 Kotlin 或者 Python 的 Lambda 表达式和高阶函数。

Lambda 表达式其实就是一个未声明的函数表达式（匿名函数）。其优势有 3 个：

- 比较轻便，即用即扔，很适合需要完成一项功能，但是此功能只在此一处使用，连名字都很随意的情况下。
- 作为匿名函数，一般用来给 filter、map 这样的函数式编程服务；
- 可以作为回调函数，传递给某些应用，比如消息处理。

而高阶函数就是把函数作为参数或者返回值的函数。

手写线程安全的单例模式代码

```
// 懒汉式
public class Single{
    private volatile static Single instance;
    private SIngles(){}
    public static Single getInstance(){
        if (instance == null){
            synchronized(Single.class){
                if (instance == null)
                    instance = new Single();
            }
        }
        return instance;
    }
}

// 内部类方案
public class Single{
    private Single(){}
    private static class Holder{
        private static Single INSTANCE = new Single();
    }
    public static Single getInstance(){
        return Hodler.INSTANCE;
    }
}
```

针对目前的简历

Handler 机制引出的 ThreadLocal，几种常用的 ThreadLocal，讲讲原理。

ThreadLocal 是一个线程内部的数据存储类，每个线程都会保存一个 ThreadLocalMap 的实例 threadlocals。ThreadLocalMap 是 ThreadLocal 的内部类，所以每个 Thread 都有一个对应的 ThreadLocalMap。里面保存了一个 Entry 数组，这个数组专门用于保存 ThreadLocal 的值。通过这样的方式，就能让我们在多个线程中互不干扰存储和修改数据。

ThreadLocal 是一个泛型类，其主要核心是 set() 和 get() 方法，在 set 方法中，我们先获取当前线程，再根据当前线程获取 ThreadLocalMap，如果该 ThreadLocalMap 不为空，就把 ThreadLocal 和我们想存放的数据设置进去。如果 ThreadLocalMap 为空的话，则先创建 ThreadLocalMap 后再设置数据。

在 get 方法中，同样是先取出当前的 ThreadLocalMap 对象，如果该对象为空，则调用 setInitialValue()，默认情况下 initialValue() 方法返回为 null，不过我们可以通过重写 initialValue() 方法来修改它。。如果取出的 ThreadLocalMap 对象不为空，那就取出它的 table 数组并找出对应的数据。

因为 ThreadLocal 的 get() 和 set() 方法操作的对象都是当前线程的 ThreadLocalMap 对象的 table 数组，它的所有读写操作都来自各自线程的内部，所以才能做到不同线程互不干扰存储和修改数据。

在 Android 中，你是如何做系统签名的?

1. 首先必须在应用的 Manifest.xml 中增加 android:sharedUserId="android.uid.system"
2. 然后打包出未签名的 apk，再编译出 signapk.jar、从系统源码目录获取 platform.x509.pem、platorm.pk8 两个文件，利用 java -jar 命令打包出带系统签名的 apk。

   > 命令为 java -jar signapk.jar platform.x509.pem platform.pk8 未签名.apk 已经签名.apk

讲讲 Android 中常用的设计模式

- 单例模式

  主要作用：省去创建一种对象创建的时间，对系统内存的使用频率降低，降低 GC 压力，缩短 GC 停顿的时间。

  最终优化版本：静态内部类。

- 建造者模式

  适用于构造一个对象需要很多个参数，并且参数的个数或者类型不固定的时候。比如我开源的图片压缩库、Android 中的 AlertDialog、Glide、OkHttp。

- 适配器模式

  在 Android 中应用：Adapter

- 装饰模式

  优点是：对于拓展一个对象的功能，装饰模式比继承更加灵活，不会导致类的个数急剧增加；可以通过一种动态的方式来拓展一个对象的功能；可以对一个对象进行多次装饰，通过使用不同的具体装饰类以及这些装饰类的排列组合。

  Android 中的应用：Context。

HTTP GET 和 POST 的区别

1. GET 的重点是从服务器上获取资源；POST 的重点是向服务器发送数据。
2. GET 传输数据是通过 URL 请求，以 field = value 的方式，这个过程对用户是可见；而 POST 传输数据通过 HTTP 的 POST 机制，将字段与对应值封存在请求体中发送给服务器，这个过程对用户是不可见的。
3. GET 的数据传输量较小，因为 URL 的长度是有限制的，但效率较高；而 POST 可以传输大量数据，所以上传文件不能用 GET 方式。
4. GET 是不安全的，因为 URL 是可见的，可能会泄漏私密信息；而 POST 较 GET 安全性更高。
5. GET 方式只能支持 ASCII 字符，向服务器传的中文字符可能会乱码；而 POST 支持标准字符集，可以正确传输中文字符。

常见的 HTTP 首部字段

- 通用的首部字段（请求报文和响应报文都会使用的首部字段）
    - Date：报文创建时间
    - Connection：连接的管理
    - Cache-Control：缓存的控制
    - Transfer-Encoding：请求报文的传输编码格式
- 请求首部字段（请求报文会使用的首部字段）
    - Host：请求资源所在服务器
    - Accept：可处理的媒体类型
    - Accept-Charset：可接受的字符集
    - Accept-Encoding：可接受的内容编码
    - Accept-Language：可接受的自然语言
- 响应首部字段（响应报文会使用的首部字段）
    - Accept-Ranges：可接受的字节范围
    - Location：令客户端重新定向的 URI
    - Server：HTTP 服务器的安装信息
- 实体首部字段（请求报文和响应报文的实体部分都会使用的首部字段）
    - Allow：资源科支持的 HTTP 方法
    - Content-Type：实体主体的类型
    - Content-Encoding：实体主体适用的编码方式
    - Content-Language：实体主体的自然语言
    - Content-Length：实体主体的字节数
    - Content-Range：实体主体的位置范围，一般用于发出部分请求时使用

Cookie 和 Session 的区别

Cookie 和 Session 都是保存客户端状态的机制，它们都是为了解决 HTTP 无状态的问题的。它们有明显的不同：

- Cookie 将状态保存在客户端；而 Session 把状态保存在服务器端。
- Cookie 不是很安全，别人可以通过分析存在本地的 Cookie 并进行 Cookie 欺骗，考虑到安全应该用 Session。
- Session 会在一定时间内存在服务器上，当访问增多，会比较占用服务器的性能，考虑到减轻服务器性能方面，应当考虑 Cookie。

介绍下这个 iSchool 协议，并讲下它是如何支持原生和 H5 交互的?

iSchool 协议是我们的 iSchool 客户端专有的协议，主要用于网页和客户端、服务器和客户端交互。协议以 iSchool:// 开头，后面更上动作命令和参数。

目前我们支持的参数包括 page 和 function。

page 作用于服务器指定客户端开启特定的页面，实际上我们 APP 的主页菜单入口就是通过接口返回 page 协议的方式来做跳转映射的。

function 协议主要用于网页 H5 和客户端双向通信专用。网页通过 ischool://function/ 协议调用客户端指定命令，客户端执行结果以回调的形式调用 JS 提供的。

目前我们的 function 协议支持是否隐藏原生标题栏、打开一个新的 URL、安装 APP、播放视频、录制视频、选择图片、支付以及分享到微信等功能；

我们会分别定义一个 IschoolFunction 和 IschoolPage 接口，暴露各种方法。然后定义一个 Ischool 协议解析器类，用于解析协议，解析后把参数存放到 Map 中，方便原生页面读取。

你在 iSchool 项目中都对分别对 6.0、7.0、8.0 做了哪些适配?

- 在 6.0 中，当时我加入公司时，项目还使用的 SdkVersion 还是 22，所以不存在权限问题，并且还在使用 HttpClient 库，所以我首先是处理了 HttpClient 在 23 版本不能直接使用的问题，主要是在 build.gradle 加上：useLibrary 'org.apache.http.legacy' 这句话。对于动态权限方面，主要是首先检查应用是否有这个权限，如果没有权限则采用 requestPermission 来请求权限。

  但在 Android 原生系统中，如果第二次弹出权限申请的对话框，会出现「以后不再弹出」的选项，当用户勾选后，再申请权限 shouldShowRequestPermissionRationale() 方法就会返回 true，导致不会弹出权限申请的对话框。我们一开始是采用弹窗告诉用户，但后面发现有些手机，比如小米4 就一直让 shouldShowRequestPermissionRationale() 方法返回 false，但也不会弹出权限申请对话框。所以我们最终的处理方案是在用户拒绝权限的回调里面，直接弹窗提示，并提供到系统设置页面进行设置的处理。
- 在 7.0 中，虽然提供了不少新特性，比如多窗口、通知增强，低电耗模式等，但我们在项目中主要适配了根据路径获取图片需要采用 FileProvider 的问题，之前我们可能都是使用 Uri.fromFile() 方法来获取文件的 Uri，但 7.0 以后必须采用 FileProvider.getUriFromFile() 方法。
- 在 Andrid 8.0 中，主要适配的是不能再直接安装 APK，而需要在 manifest.xml 文件中增加 REQUEST_INSTALL_PACKAGES 权限。

你是怎么处理各种机型的屏幕适配的?

在我们的项目中，对 Android 的各种机型做适配可谓是花了大工夫。

1. 首先 Android 提供了 dp 方案，这也是我们最开始的适配方案，但我们很快发现了问题。我们的应用里面有个 banner 图轮播的功能，我们高度采用 dp 做处理，宽度采用 match_parent 占满屏幕宽度，对大多数手机显示都不存在问题。但我们的屏幕宽度其实是不一样的，而高度固定后肯定会出现图片显示不完全的情况。
2. 所以我们采用了一个笨办法，那就是宽高限定符适配，也就是穷举市场面主流手机的宽高像素点，但很快发现这样维护起来相当麻烦，所以一直在寻求好办法进行处理。
3. 后面发现了鸿洋的 AutoLayout，简单看了下里面的源码，也是受宽高限定符的影响，我们可以直接在布局中写 px 值，这样布局会等比例放缩，但由于它不能兼顾到自定义 View 以及存在性

能问题而且他本人也没有维护的情况下，所以我们后面并没有用这个方案。
4. 终于在最后今日头条开源了一个比较不错的适配方案。我们直接通过修改 density 的值，强行把所有不同尺寸分辨率的手机的宽度值 dp 改成一个统一的值，这就解决了所有的适配问题。

设计模式的原则

设计模式可以让我们的程序更健壮、更稳定、更加容易拓展，编写的时候我们需要遵循 6 大原则：

1. 单一职责原则：

   单一原则很简单，就是将一组相关性很高的函数、数据封装到一个类中，换句话说，一个类应该有职责单一。

2. 开闭原则：

   开闭原则就是说一个类应该对于拓展是开放的，但是对于修改是封闭的。因为开放的 APP 或者是系统中，经常需要升级和维护等，一旦进行修改，就容易破坏原有的系统，甚至带阿里一些难以发现的 Bug。所以开闭原则相当重要。

3. 里氏替换原则：

   里氏替换原则的定义为：所有引用基类的地方必须能透明地使用其子类对象。简单地说，就是以父类的形式声明的变量或形参，赋值为任何继承于这个父类的之类不影响程序的执行。

4. 依赖倒置原则：

   依赖倒置主要是实现解耦，使得高层次的模块不依赖于低层次模块的具体实现细节。几个关键点：

   - 高层模块不应该依赖底层模块。二者都应该依赖其抽象类或接口；
   - 抽象类或接口不应该依赖于实现类；
   - 实现类应该依赖于抽象类或接口；

5. 接口隔离原则：

   接口隔离原则定义：类之间的依赖关系应该建立在最小的接口上。其原则是将非常庞大的、臃肿的接口拆分成更小的更具体的接口。

6. 迪米特原则：

   描述的原则：一个对象应该对其他的对象有最少的了解。什么意思呢？就是说一个类应该对自己调用的类知道的最少。还是不懂？其实简单来说：假设类 A 实现了某个功能，类B需要调用类 A 的去执行这个功能，那么类 A 应该只暴露一个函数给类 B，这个函数表示是实现这个功能的函数，而不是让类 A 把实现这个功能的所有细分的函数暴露给 B。

Android 中常用的设计模式

1. 单例 => 调用系统服务时拿的 Binder 对象；
2. Builder

   主要是为了把一个复杂对象的构造和它的表示分离，使得同样的构造过程可以创建不同的表示。

   => 常见的比如 Android 的 Dialog。

   > 其中我的图片压缩库就运用了单例和 Builder 模式。

3. 工厂方法模式

   根据传入的参数决定创建哪个对象。

=> 典型的获取系统服务 `getSystemService()`。

4. 策略模式

   如何使用策略模式呢，我不打算写示例代码了，简单描述一下，就将前面说的算法选择进行描述。我们可以定义一个算法抽象类 AbstractAlgorithm，这个类定义一个抽象方法 sort()。每个具体的排序算法去继承 AbstractAlgorithm 类并重写 sort() 实现排序。在需要使用排序的类 Client 类中，添加一个 setAlgorithm(AbstractAlgorithm al)；方法将算法设置进去，每次 Client 需要排序而是就调用 al.sort()。

   => 比如 Android 的属性动画的使用插值器。

5. 责任链模式

   使多个对象都有机会处理请求，从而避免请求的发送者和接收者直接的耦合关系，将这些对象连成一条链，并沿这条链传递该请求，直到有对象处理它位置。

   => 比如 Android 的事件分发机制。

6. 观察者模式

   => 比如 Adapter 的 notifyDataSetChanged()。

7. 迭代器模式

   => Java 的迭代器 Iterator 类

8. 代理模式

   为其他类提供一种代理以控制这个对象的访问。

   => Android 中的典型应用就是 AIDL 的生成，它根据当前的线程判断是否要跨进程访问，如果不需要跨进程访问就直接返回实例，如果需要跨进程则返回一个代理。在跨进程通信时，需要把参数写入到 Parcelable 对象，然后再执行 transact() 函数，我们要写的代码挺多的。AIDL 通过生成一个代理类，代理类中自动帮我们写好这些操作。

9. 适配器模式

   => 典型的 Android 的 ListView 和 RecyclerView。它们只关心自己的 ItemView，而不用关心这个 ItemView 里面具体显示的是什么。而数据源存放的是显示的内容，它保存了 ItemView 要显示的内容。ListView 和数据源本身没有关系，这时候，适配器提供了 getView() 方法给 ListView 使用，每次 ListView 只需要提供位置信息给 getView() 函数，然后 getView() 函数根据位置信息向数据源获取对应的数据，根据数据返回不同的 View。

10. 装饰模式

```
public abstract class Component{
    public abstract void operate();
}
public class ConcreteComponent extends Component{
    public void operate(){
        //具体的实现
    }
}
public class Decorator{
    private Component component;
    public Decorator(Component component){
        this.component=component;
```

```
    }
    public void operate(){
        operateA();
        component.operate();
        operateB();
    }
    public void operateA(){
        //具体操作
    }
    public void operateB(){
        //具体操作
    }
}
```

=> 在 Android 中的 Context 。

11. 享元模式

    使用享元对象有效地支持大量的细粒度对象。

    => 比如 Java 的常量池，线程池等，主要是为了重用对象。在 Android 里面的 Message.obtain() 就是一个很好的例子。

讲讲 LeakCanary 的实现原理。

LeakCanary 就是通过注册 `ActivityLifecycleCallbacks` ，监听生命周期方法的回调，作为整个内存泄漏分析的入口。每次 `onActivityDestroyed(Activity activity)` 方法被回调之后,都会创建一个 KeyedWeakReference 对相应的 Activity 的状态进行跟踪。

1. 在后台线程（HandlerThread）检查引用是否被清除，如果没有，手动调用 GC。
2. 如果引用还是未被清除，把 heap 内存 dump 到 APP 对应的文件系统中的一个 `.hprof` 文件中。
3. 在另外一个进程中的 `HeapAnalyzerService` 有一个 `HeapAnalyzer` 使用 HAHA 开源库解析这个文件。
4. 得益于唯一的 reference key, `HeapAnalyzer` 找到 `KeyedWeakReference`，定位内存泄漏。
5. `HeapAnalyzer` 计算到 GC roots 的最短强引用路径，并确定是否是泄漏。如果是的话，建立导致泄漏的引用链。
6. 引用链传递到 APP 进程中的 `DisplayLeakService`， 并以通知的形式展示出来。

多渠道打包？美团的多渠道打包？

之前有对美团的多渠道打包方案进行调研。

美团之前的多渠道打包方案主要是在 META-INF 目录下添加空文件，用空文件的名称来作为渠道的唯一标示，之前在 META-INF 目录下添加文件是不需要重新签名应用的，所以好使。但 Android 7.0 出了新的签名方式，把 META-INF 列入了保护区，向 META-INF 目录添加空文件将会对其他区产生影响，所以美团之前的多渠道打包方式将会出现问题。但美团响应速度很快，很快就出现了 Walle，可以完美在新签名方式下进行多渠道打包。

新的签名方案主要是在不受保护的 APK Signing Block 上做文章。好像是在 APK 中添加 ID-value。

我们的 iSchool 项目采用多渠道打包并不是为了统计各个商店的下载量等，而是主要为了应对我们的商业模式。所以采用的传统的方式进行多渠道打包，在 Gradle 文件里面通过 productFlavors 标签设置不同的 applicationId 和其它信息。并且设置 task 做到指定打一个代理商的渠道包。我们对资源文件也是不一样的。我们对同一个资源采用不一样的后缀名称，然后通过字符串截取去判断真正的资源文件，达到不一样的项目引用不一样的资源的目的。

RxJava 源码。

简单看了一下 RxJava2 的 源码订阅过程。

- **创建 Observable & 定义需要发送的事件**

  创建 Observable 的本质其实就是创建了一个 ObservableCreate 类的对象。它的内部有 1 个 subscribeActual() 方法。

- **创建 Observer & 定义响应事件的行为**

  Observer 是一个接口，内部定义了 4 个接口方法，分别用于响应被观察者发送的不同事件。

- **通过订阅（subscribe）连接观察者和被观察者**

  订阅的本质实际上就是调用 ObservableCreate 类对象的 subscribeActual()。

  执行逻辑为：

  1. 创建一个 CreateEmitter 对象（封装成 1 个 Disposable 对象）；
  2. 调用被观察者 ObservableOnSubscribe 对象复写的 subscribe()；
  3. 调用观察者 Observer 复写的 onSubscribe()；
  4. 被观察者内部的 subscribe() 依次回调 Observer 复写的方法。

RxJava 内置了许多操作符，每个操作符的 subscribeActual() 中的默认实现都不相同。但基本原理都是封装传入的 Observer 对象，再调用 Observable 复写的 subscribe()。

各种操作符内的 Observable 中的 SubscirbeActual() 方法都含有一句代码：source.subscribe(Observer)，这个 source 就是上游创建的被观察者对象 Observable。所以才导致 RxJava 的多个被观察者 Observable 发送事件的顺序是自上而下的，和代码顺序一样。

OkHttp 源码。

OkHttp 支持异步请求和同步请求。因为异步请求不会阻塞主线程，所以我们通常会直接把异步请求写在 UI 主线程中。

我们首先会创建一个 OkHttpClient 对象，在源码中可以知道，OkHttpClient 对象是通过 Builder 模式创建的，在 OkHttpClient.Builder() 类的构造方法中，对它的属性都有默认值和默认对象。

创建好 OkHttpClient 对象之后，我们会通过调用它的 newCall(Request request) 方法来创建一个 RealCall 对象。

对于异步请求，我们会直接调用 RealCall 的 enqueue() 方法，这个方法里面又会调用调度器对象 Dispatcher 的 enqueue(AsyncCall call) 方法。这个方法接收一个 AsyncCall 对象参数。

从 `Dispatcher` 的 `enqueue(AsyncCall call)` 方法中可以看到，得到异步任务之后，如果异步任务运行队列中的个数小于 64 并且每个主机正在运行的异步任务小于 5，则将该异步任务加入到异步运行队列中，并通过线程池执行该异步任务，若不满足以上两个条件，则将该异步任务加入到预备异步任务队列中。

AsyncCall 是 RealCall 的内部类。它主要是响应网络请求的各种回调，并最后从 Dispatcher 对象的异步请求队列中移除该任务，并将符合条件的预备异步任务队列中的任务加入到正在运行的异步任务队列中，并将其放入线程池中执行。

在 AsyncCall 中最重要的就是 execute() 方法。这个方法里面会通过一个方法把所有的拦截器做聚合之后生成一个拦截器调用链对象并返回。

对于同步请求，和异步不一样的是在直接执行 RealCall 对象的 execute() 方法。它会调用 Dispatcher 对象的 executed(RealCall call) 方法将这个同步任务加入到 Dispatcher 对象的同步任务队列中去。接着会调用 getResponseWithInterceptorChain() 方法获取到请求的响应对象。最终，也是在 finally 中将同步任务从 Dispatcher 对象的同步任务队列中移除。

拦截器是 OkHttp 设计的精髓所在，每个拦截器负责不同的功能，使用责任链模式，通过链式调用执行所有的拦截器对象中的 intercept() 方法。拦截器在某种程度上也借鉴了网络协议中的分层思想，请求时从最上层到最下层，响应时从最下层到最上层。

系统中已经默认添加了重试重定向、桥接拦、缓存、连接服务器、请求服务器等拦截器，但我们依然可以自定义拦截器。

编写起来也非常简单，我们只需要实现 Interceptor 接口，并重写其中的 intercept() 方法即可。

ART 和 Dalvik 虚拟机以及 JVM 的区别

首先看看 JVM 和 Dalvik 的区别。

- Dalvik 基于寄存器，而 JVM 是基于栈的。
- Dalvik 运行 dex 文件，而 JVM 运行 Java 字节码。

再来看看 ART 和 Dalvik。

ART 和 Dalvik 的机制不同。在 Dalvik 下，应用每次运行的时候，字节码都需要通过即时编译器转换为机器码，这会拖慢应用的运行效率，而在 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。这个过程叫做预编译。这样的话，应用的启动和执行都变得更加快速。

ART 拥有着给带来系统性能的显著提升，让应用启动更快、运行更快、体验更流畅、触摸反馈更及时、更长的电池续航能力、支持更低的硬件等优势。当然 ART 也有缺点，那就是机器码占用的存储空间更大，而且应用的安装时间会变长。

sleep 和 wait 有什么区别

功能差不多，都用来做线程控制，但 sleep 不会释放同步锁，wait() 会释放同步锁。

用法不同，sleep 可以用时间指定来使它自动醒过来，如果时间不到，我们只能通过调用 interreput() 来强行打断，而 wait() 可以用 notify() 直接唤起。

讲讲目前 Android 主要的热修复方案

总的来说热修复主要是利用三个方法：类加载、底层替换和 Instant Run。

采用类加载方案的目前以腾讯系为主，比如 Tinker 等，但这个方案需要重启 APP 后让 ClassLoader 重新加载新的类。因为类是无法被卸载的，要想重新加载新的类就需要重启 APP。所以采用类加载方案的热修复框架是不能即时生效的。

而底层替换方案不会加载新类，而是直接在 Native 层修改原有的类，由于是在原有类进行修改，所以会有比较多的限制，不能增减原有的方法和字段。但这样的方法可以立即生效并且不需要重启，目前采用底层替换方案的主要以阿里系为主。比如阿里百川、AndFix 等。**底层替换方案存在一个问题，那就是需要针对 Dalvik 虚拟机和 ART 虚拟机做适配没需要考虑指令集的兼容问题，需要 native 代码支持，兼容性也会有一定的影响。**

除了资源修复，当然还可以借鉴 Instant Run 的原理。比如美团开源的 Robust。简单看了下 Robust 插件的原理，它主要是对每个产品代码的每个函数都在编译打包的阶段自动地插入一段代码，插入过程对业务开发是完全透明的。Robust 会为每个 class 增加一个类型为 ChangeQuickRedirect 的静态成员，而在每个方法前都插入了使用 changeQuickRedirect 相关的逻辑，当 changeQuickRedirect 不为 null 的时候，可能会执行到 accessDispatch 从而替换掉之前老的逻辑，达到 fix 的目的。

讲讲 HTTPS 的加密过程。

客户端请求 HTTPS 连接 => 服务器返回证书（证书中包含公钥）=> 客户端产生密钥并用服务器的公钥加密 => 客户端发送加密密钥 => 服务器返回加密的密文通信

AsyncTask 的原理

AsyncTask 内部封装了两个线程池和一个 Handler。

其中一个线程池负责任务调度，这个线程池实际上是一个静态内部类，即所有 AsyncTask 对象公有的；其内部维护了一个双向队列，容量根据元素数量调节；通过同步锁修饰 execute()，保证 AsyncTask 的任务是串行执行；每次都是从队列头部取出任务。

另一个线程负责真正执行具体的线程任务。实际上就是一个已经配置好的可执行并行任务的线程池。

而 Handler 主要负责异步通信和消息传递。

OkHttp 的优势。

1. 对同一个主机发出的所有请求都可以共享相同的套接字连接；
2. 使用线程池来复用连接以提高效率；
3. 提供了对 gzip 的默认支持来降低传输内容的大小；
4. 对 HTTP 响应的缓存机制，可以避免不必要的网络请求；
5. 当网络出问题时，OkHttp 会自动重试一个主机的多个 IP 地址；

Android 9.0 新特性

- 全面屏支持，刘海屏手机可能会成为趋势；
- 通知栏多种通知；
- 多摄像头的更多动画；
- GPS 定位外的 WIFI 定位；
- 网络还有神经网络;
- Material Design 2.0;
- Android Dashboard：用户可以自己看清楚自己在手机上都做了什么，在哪个应用停留了多少时间，停留过长还会有提示；
- 增加夜间模式；
- Adavance Battery：需要用户手动开启该模式，机器学习，降低后台占用，提升续航；
- Shush：屏幕朝下完全进入勿扰状态。除了闹钟，其他自动调整为静音或者震动的模式。
- Actions 和 Slices：检测用户行为，让系统做出对应的动作。

HR 猜测面

你对未来有什么规划?

你过往工作中取得过怎样的成功，又遭遇了哪些失败?

咕咚总监面自我介绍

面试官您好，我是刘世麟，非常荣幸能参加咕咚的复试，下面我简单介绍一下我的个人情况：我从实习到现在一直在致学教育工作，从事 Android 开发，凭借良好的工作能力和沟通能力，连续两年蝉联「优秀员工」称号，在今年初被公司内聘为技术总监助理，协助技术总监开展部门管理和项目推动工作。在工作之外，我喜欢编写技术博客和在 GitHub 上贡献开源代码，目前在 GitHub 上总共拥有 7k 左右的 Star，数篇技术博客也有数十万阅读。我非常地热爱移动开发，我希望我的技术能像我的发际线一样深入。不瞒您说，我一直认为咕咚是一家技术氛围良好的企业，所以一直渴望加入，通过前面和徐哥以及超哥的交流，我更加地坚定了我内心的想法。同样，我认为我很适合咕咚的技术团队。首先，我非常的热爱移动开发，而且学习能力较强，这和咕咚目前的团队配置非常相符，所以我认为我可以很快融入团队，并迅速开启开发工作。此外，咕咚目前采用的 Java 和 Kotlin 混合开发 以及 MVVM 架构都是我可以迅速上手的。周三和徐哥交流的咕咚项目的问题点，我也专门去做了探索。听闻咕咚最近在筹备团队技术博客的事情，我希望并且相信我能在这里发挥好自己的优势!

今日头条面试

1. ClassLoader 加载原理。

   ### ClassLoader加载类的原理

   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 `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。

2. LeakCanary 工作原理，GC 如何回收，可以作为GC 根结点 的对象有哪些?

   JVM垃圾回收的根对象的范围有以下几种：

   （1）虚拟机（JVM）栈中引用对象 （2）方法区中的类静态属性引用对象

   （3）方法区中常量引用的对象（final 的常量值）

   （4）本地方法栈JNI的引用对象
3. JVM 内存模型，如何理解 Java 虚函数表
4. 如何从 100 万个数中找到最小的一百个数，考虑算法的时间复杂度和空间复杂度。
5. JS 和 Java Native 如何通信?
6. APP 加固怎么做?
7. MVP 和 MVC 以及 MVVM 的主要却别，为什么 MVP 要比 MVC 好?
8. Android 的混淆原理?
9. 如何设计一个安卓的画图库，做到对扩展开放，对修改封闭，同时又保持独立性
10. 怎样做系统调度?
11. 数组实现队列。
12. 实现 LRUCache。
13. HashMap 原理，Hash 碰撞，如何设计让遍历效率更高?
14. ConcurrentHashMap 特点是什么？如何实现的。JDK 1.8 用红黑树实现，什么是红黑树?
15. Syncronized 和 Lock 的区别，join 和 yield 的用法，volitate 用法，CAS 怎么实现的?
16. jvm内存结构 手写代理模式 手写线程安全的单例模式；GC算法有哪些？Android触摸机制；除了读取trace文件，还能如何获取到ANR异常?
17. 一个行列都有序的数组，给定一个数，找出具体的位置。

18. Retrofit 原理。

    主要是通过动态代理将接口直接转换成代理对象。动态代理和静态代理的区别，动态代理直接在虚拟机层面构建字节码对象。

19. View自定义的流程，实现哪些方法。
20. 链表回文结构。
21. Android内存优化怎么优化。性能优化怎么优化。卡顿是什么原理。
22. 中序遍历，非递归实现