# Android 面试指南

# Android 面试指南

### 关于

受 [android-interview-questions](https://github.com/MindorksOpenSource/android-interview-questions) 项目启发，这里想发挥众多 Android 中国开发者的力量，整理一份高质量、范围全的 Android 面试指南，旨在帮助更多的 Android 开发者提升技术，找到工作。

现在还是项目初期，项目背景见这里：[想跟大家一起做件小事](http://mp.weixin.qq.com/s/t038R0bDDZ6dg4bwDoj2cQ)，也欢迎关注微信公众号 stormzhang，后续有任何进展我都会在公号进行通知的。

目前该项目有如下初步的计划：

1. 翻译 [android-interview-questions](https://github.com/MindorksOpenSource/android-interview-questions) 项目，不过翻译仅是我们的第一步而已，这个比较简单，目前第一步翻译工作已经被认领完毕；
2. 因为原项目很多只是列了一些问题，但是都没有答案的，所以第二步我们就是认领问题，整理答案，务必保证高质量、易理解，因为问题很多，所以这一步需要花费不少精力，也需要更多的同学参与进来，目前还在第一步阶段；
3. 第三步是补充与完善，原项目虽然列了不少领域，但是总归有些遗漏的，比如 Android 安全、插件化、Kotlin 等等，第三步是找在某一领域研究比较深的同学加入进来，对一些领域进行补充、完善，甚至会推出一些专题等；

## Contents

* [数据结构和算法](#数据结构与算法)
* [Java 核心](#java-核心)
* [Android 核心](#android-核心)
* [架构](#架构)
* [设计问题](#设计问题)
* [工具和技能](#工具和技能)
* [Android 测试驱动开发](#android-测试驱动开发)
* [其他](#其他)

### 数据结构与算法

数据结构与算法问题的难度完全取决于你所申请的公司

* 数组
  + 数组由一组相同的数据类型组成。它存储在连续的内存空间内，使用索引可以找到元素的地址。数组包括一维数组和多维数组,一维数组是最简单的数据结构,也是最常用的。

|  |  |  |
| --- | --- | --- |
| * + 算法 | * + 平均 | * + 最坏 |
| * + 空间（Space） | * + O(n) | * + O(n) |
| * + 查找（Search） | * + O(n) | * + O(n) |
| * + 插入（Insert） | * + O(n) | * + O(n) |
| * + 删除（Delete） | * + O(n) | * + O(n) |

* 链表
* 链表看起来更像树，而不是数组，它使用一组结点来表示一个序列。每一个结点都包含数据和一个指针。在链表中，结点中的数据可以为任意类型，而指针则是指向下一结点的引用。链表包含一个头结点和一个尾结点。头结点是链表中的第一个结点，尾结点是最后一个结点。链表不是一个循环数据结构，所以尾结点没有指向头结点的指针，指针为空。一些基础方法的时间复杂度如下：
* | 算法 | 平均 | 最坏 |  
  |:------------:|:----:|:----:|  
  | 空间 (Space) | O(n) | O(n) |  
  | 查找 (Search) | O(n) | O(n) |  
  | 插入 (Insert) | O(1) | O(1) |  
  | 删除 (Delete) | O(1) | O(1) |
* 双向链表
* 一个双向链表首先是一个链表，但是在每个结点中有两个指针，前驱指针指向前驱结点，后继指针指向后继结点。双向链表也有一个头结点，头结点的后继指针指向第一个结点。最后一个结点的后继指针指向空，但是如果最后一个结点的后继指针指向第一个结点，这时称这个链表为双向循环链表。双向循环链表能非常方便地从每个结点查找它的前驱结点和后继结点。
* 双向链表
* 双向链表
* | 算法 | 平均 | 最坏 |  
  |:------------:|:----:|:----:|  
  | 空间 (Space) | O(n) | O(n) |  
  | 查找 (Search) | O(n) | O(n) |  
  | 插入 (Insert) | O(1) | O(1) |  
  | 删除 (Delete) | O(1) | O(1) |
* 栈
  + 栈是一个有着「后进先出」特性的基础数据结构，这就意味着最后一个入栈的元素，也是第一个出栈的。栈就像是一堆书，想要得到书堆中的第一本书（最下面一本），必须把其他的书都先拿走。向栈中添加一个元素的操作被称为 Push（入栈），删除一个元素的操作被称为 Pop（出栈），查看且不删除最后一个入栈的元素的操作被称为 Top 。[实现栈的常用方法是使用链表（LinkedList），也可以使用不允许空值的 StackArray（使用数组实现），还有允许空值的 Vector](https://en.wikibooks.org/wiki/Data_Structures/Stacks_and_Queues#Performance_Analysis)
  + <tr>  
     <th>算法</th>  
     <th>平均</th>  
     <th>最坏</th>  
     <th>图形表示</th>  
    </tr>  
    <tr>  
     <td>空间 (Space)</td>  
     <td>O(n)</td>  
     <td>O(n)</td>  
     <td rowspan="5">  
     <img src="https://upload.wikimedia.org/wikipedia/commons/thumb/2/29/Data\_stack.svg/250px-Data\_stack.svg.png"/>  
     </td>  
    </tr>  
    <tr>  
     <td>查找 (Search)</td>  
     <td>O(n)</td>  
     <td>O(n)</td>  
    </tr>  
    <tr>  
     <td>入栈 (Push)</td>  
     <td>O(1)</td>  
     <td>O(1)</td>  
    </tr>  
    <tr>  
     <td>出栈 (Pop)</td>  
     <td>O(1)</td>  
     <td>O(1)</td>  
    </tr>  
    <tr>  
     <td>查看栈顶 (Top)</td>  
     <td>O(1)</td>  
     <td>O(1)</td>  
    </tr>
* 队列
* 优先队列
* 动态编程
* 字符串操作
* 二叉树
* 二叉搜索树
* 排序算法
* 哈希表与哈希图
* 广度优先搜索
* 深度优先搜索
* 贪心算法

所谓贪心算法(贪婪算法)，是在求解一个问题时，作出当前最好的选择，而不考虑大局。也就是说，这种算法的每一种实现，只是在作出一个某种方面上的局部最优解。

我们可以说贪心算法是「短视的」， 「不可恢复」的

此算法没有固定框架，只有每个人贪心策略的选择，而导致的不同姿态的代码实现。

来自wikipedia：

贪心法，又称貪心演算法、貪婪演算法、或稱貪婪法，是一种在每一步选择中都采取在当前状态下最好或最优（即最有利）的选择，从而希望导致结果是最好或最优的算法。[1]比如在旅行推销员问题中，如果旅行员每次都选择最近的城市，那这就是一种贪心算法。

贪心算法在有最优子结构的问题中尤为有效。最优子结构的意思是局部最优解能决定全局最优解。简单地说，问题能够分解成子问题来解决，子问题的最优解能递推到最终问题的最优解。

贪心算法与动态规划的不同在于它对每个子问题的解决方案都做出选择，不能回退。动态规划则会保存以前的运算结果，并根据以前的结果对当前进行选择，有回退功能。

贪心法可以解决一些最优化问题，如：求图中的最小生成树、求哈夫曼编码……对于其他问题，贪心法一般不能得到我们所要求的答案。一旦一个问题可以通过贪心法来解决，那么贪心法一般是解决这个问题的最好办法。由于贪心法的高效性以及其所求得的答案比较接近最优结果，贪心法也可以用作辅助算法或者直接解决一些要求结果不特别精确的问题。

### Java 核心

* 解释一下 OOP 的概念
* 面向对象编程是使用类，对象，[继承性](https://zh.wikipedia.org/wiki/%E7%BB%A7%E6%89%BF_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6))，[多态性](https://zh.wikipedia.org/wiki/%E5%A4%9A%E5%9E%8B_(%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6))，[封装性](https://zh.wikipedia.org/wiki/%E5%B0%81%E8%A3%9D_(%E7%89%A9%E4%BB%B6%E5%B0%8E%E5%90%91%E7%A8%8B%E5%BC%8F%E8%A8%AD%E8%A8%88))和[抽象](https://zh.wikipedia.org/wiki/%E6%8A%BD%E8%B1%A1%E5%8C%96_(%E8%A8%88%E7%AE%97%E6%A9%9F%E7%A7%91%E5%AD%B8))的一种程序设计方法。。

1. 抽象

在面向对象的概念中，所有对象都是由类来描述，但是反过来，并不是所有类都是用来描述对象的。如果一个类中没有包含足够信息来描绘一个具体的对象，这样的类就是抽象类。

1. 继承

继承（英语：inheritance）是面向对象软件技术当中的一个概念。如果一个类别A“继承自”另一个类别B，就把这个A称为“B的子类别”，而把B称为“A的父类别”也可以称“B是A的超类”。继承可以使得子类别具有父类别的各种属性和方法，而不需要再次编写相同的代码。在令子类别继承父类别的同时，可以重新定义某些属性，并重写某些方法，即覆盖父类别的原有属性和方法，使其获得与父类别不同的功能。另外，为子类别追加新的属性和方法也是常见的做法。 一般静态的面向对象编程语言，继承属于静态的，意即在子类别的行为在编译期就已经决定，无法在执行期扩充。 有些编程语言支持多重继承，即一个子类别可以同时有多个父类别，比如C++编程语言；而在有些编程语言中，一个子类别只能继承自一个父类别，比如Java编程语言，这时可以利用接口来实现与多重继承相似的效果。 现今面向对象程式设计技巧中，继承并非以继承类别的“行为”为主，而是继承类别的“型态”，使得元件的型态一致。另外在设计模式中提到一个守则，“多用合成，少用继承”，此守则也是用来处理继承无法在执行期动态扩充行为的遗憾。

1. 封装

从字面上理解就是包装的意思，是指利用抽象数据类型，将数据和关于数据的操作封装起来，使其成为一个不可分割的独立实体。数据将会被保护在抽象数据类型的内部，仅能够通过暴露在表面的操作（public方法，比如setter和getter）来与这个对象进行交流和交互。用户不知道对象的内部细节，但是通过该对象提供的接口来访问对象。其好处是：减少耦合，方便地在未来修改调整自己，更加有把握地（精确地）控制成员，隐藏信息，实现细节。

1. 多态

使用相同的消息，使得类作出不同的反应（继承为我们使用多态打下了基础）。Java实现多态有三个必要条件：继承、重写、向上转型。

* 面向对象的三个基本元素和五个原则

1. 三个基本元素：

* 封装： 封装是把过程和数据包围起来，对数据的访问只能通过已定义的界面。面向对象计算始于这个基本概念，即现实世界可以被描绘成一系列完全自治、封装的对象，这些对象通过一个受保护的接口访问其他对象。
* 继承： 继承是一种联结类的层次模型，并且允许和鼓励类的重用，它提供了一种明确表述共性的方法。对象的一个新类可以从现有的类中派生，这个过程称为类继承。新类继承了原始类的特性，新类称为原始类的派生类（子类），而原始类称为新类的基类（父类）。派生类可以从它的基类那里继承方法和实例变量，并且类可以修改或增加新的方法使之更适合特殊的需要。
* 多态： 多态性是指允许不同类的对象对同一消息作出 响应。多态性包括参数化多态性和包含多态性。多态性语言具有灵活、抽象、行为共享、代码共享的优势，很好的解决了应用程序函数同名问题。C++中的虚函数的作用主要是实现了多态的机制。关于多态，简而言之就是用父类型别的指针指向其子类的实例，然后通过父类的指针调用实际子类的成员函数。这种技术可以让父类的指针有“多种形态”，这是一种泛型技术。所谓泛型技术，说白了就是试图使用不变的代码来实现可变的算法。比如：模板技术，RTTI技术，虚函数技术，要么是试图做到在编译时决议，要么试图做到运行时决议。

1. 五个基本原则：

* 单一职责原则（Single-Resposibility Principle）：一个类，最好只做一件事，只有一个引起它的变化。单一职责原则可以看做是低耦合、高内聚在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。
* 开放封闭原则（Open-Closed principle）：软件实体应该是可扩展的，而不可修改的。也就是，对扩展开放，对修改封闭的。
* Liskov替换原则（Liskov-Substituion Principle）：子类必须能够替换其基类。这一思想体现为对继承机制的约束规范，只有子类能够替换基类时，才能保证系统在运行期内识别子类，这是保证继承复用的基础。
* 依赖倒置原则（Dependecy-Inversion Principle）：依赖于抽象。具体而言就是高层模块不依赖于底层模块，二者都同依赖于抽象；抽象不依赖于具体，具体依赖于抽象。
* 接口隔离原则（Interface-Segregation Principle）：使用多个小的专门的接口，而不要使用一个大的总接口。
* 抽象类和接口的区别？[Link](https://arjun-sna.github.io/java/2017/02/02/abstractvsinterface/)
* 抽象类是一个可同时包含具体方法和抽象方法(方法未被实现)的类。抽象方法必须被该抽象类的子类实现。抽象类是可以继承的。
* 接口像是描述类的一张蓝图或者说是类的一种契约，它包含了许多空方法，这代表着它的所有的子类都应该拥有共同点。它的子类应该提供这些空方法的具体实现。一 个类需要用 implements 来实现接口，接口可以用 extends 来继承其他接口。
* 在设计层级理解他们的不同：

1. 抽象的层次不同：抽象类对类的整体（包括属性，行为）都可以进行抽象，接口对类的局部进行抽象，具体来说接口仅仅是对类的行为进行抽象。
2. 跨域不同：抽象类是 从各种子类中提取相似的部分，然后泛化成抽象类，子类可以继承这样的抽象类。 实现接口是 不存在is-a的关系的类们，你不可以称同样可以飞行的飞机和鸟为同一个抽象类，但是他们可以有同样的接口fly-able。抽象类的父类和派生类在概念上一致，接口的原生类和派生类在仅仅在局部行为上一致。
3. 设计层次不同：抽象类是从一堆在底层的子类们来进行抽象提取，从下往上，从而产生抽象类；接口是在直接定义的高度来声明的，然后从这个高度上往下实现此接口。抽象类是自底向上抽象而来的，接口是自顶向下设计出来的。

* 序列化是什么?如何实现它?
* 序列化是一种将对象转换为字节流的过程，目的是为了将该对象存储到内存中，等后面再次构建该对象时可以获取到该对象先前的状态及数据信息。Java中，有两种方式可以实现序列化，既可以实现Serializable接口，也可以实现Parcelable接口。然而，在Android中，我们不应该使用Serializable接口。因为Serializable接口使用了反射机制，这个过程相对缓慢，而且往往会产生出很多临时对象，这样可能会触发垃圾回收器频繁地进行垃圾回收。相比而言，Parcelable接口比Serializable接口效率更高，性能方面要高出10x多倍。
* 什么是单例？
* 单例模式指的是一个类只能被初始化一次，即只有一个实例。[单例模式限定一个类只能拥有一个实例。这在系统中只需要一个实例来和其他模块协调工作时是很实用的。单例普遍使用在只需要一个或是限制一定数量实例的系统中。（wikipedia）](https://en.wikipedia.org/wiki/Singleton_pattern)
* 什么是匿名内部类？
* 普通的类可以自然地实例化他自己，相反地，内部类是这样的类： *一定要绑定上一个外部类才能进行实例化的类*。
* 内部类提供了一种模拟车和车轮的机制，车是外部类，车轮是内部类
* 而匿名内部类也就是没有名字的内部类，因为没有名字，所以只能使用一次。
* 使用匿名内部类还有个前提条件：必须继承一个父类或实现一个接口
* 详情可见：
* http://www.cnblogs.com/nerxious/archive/2013/01/25/2876489.html
* 对字符串进行 == 和 equals() 操作时有什么区别？
* == 比较两个字符串的地址，初学者很经常拿来比较其内容，将会导致出现不等的情况。 equals()是String这个类重写的一个方法，平常的类的equals()也仅仅是比较两个变量的地址，而String类的equals()重写后，将依次比较其串中的字符。
* hashCode() 和 equals() 何时使用？
* 一般是在想要人性化地（而不是计算机式地,比较地址那样）比较两个对象的时候，我们需要使用这两个方法，或者说我们要重写这两个方法，而且有如下的原则：
  1. hashCode的存在主要是用于查找的快捷性，如Hashtable，HashMap等，hashCode是用来在散列存储结构中确定对象的存储地址的；
  2. 如果两个对象相同，就是适用于equals(java.lang.Object) 方法，那么这两个对象的hashCode一定要相同；
  3. 如果对象的equals方法被重写，那么对象的hashCode也尽量重写，并且产生hashCode使用的对象，一定要和equals方法中使用的一致，否则就会违反上面提到的第2点；
  4. 两个对象的hashCode相同，并不一定表示两个对象就相同，也就是不一定适用于equals(java.lang.Object) 方法，只能够说明这两个对象在散列存储结构中，如Hashtable，他们“存放在同一个篮子里”。
* hashCode请参考：http://blog.csdn.net/fenglibing/article/details/8905007
* Java 中的 final, finally 和 finalize?
  1. final: 修饰变量，方法，类； 修饰变量时表明这对象的值不可变，你不能为这个变量赋一个新的值，或者这样说：对基本类型而言，你不能改变其数值，对于引用，你不能将其指向一个新的引用（而引用自身是可以改变的）。 修饰方法时表明我们希望把这个方法锁定，以防止任何继承类修改它的含义，这样会确保在继承中，我们的final方法的行为不会改变，并且不会被覆盖。使用final方法的另一个考虑是效率问题：在Java早期的时候，遇到final方法，编译器会将此方法调用转为内嵌调用，如此一来以减小方法调用产生的开销。 修饰类的时候表明你不打算继承该类，而且也不允许别人这样做。
  2. finally:是异常处理中进行收场处理的代码块，比如关闭一个数据库连接，清理一些资源占用的问题。不管有没有异常被捕获，finally子句中的代码都会被执行。
  3. finalize:finalize出现的原因在于： 我们一定需要进行清理动作。Java没有用于释放对象的，如同C++里的delete调用，的方法，而是使用垃圾回收器（GC）帮助我们释放空间。当垃圾回收器准备释放对象占用的存储空间的时候，将首先调用其finalize()方法。
* 什么是内存泄露，Java 是如何处理它的？

   总的来说是：保留下来却永远不再使用的对象引用    它包括： 1. 静态集合类引起内存泄漏， 2. 当集合里面的对象属性被修改后，再调用remove()方法时不起作用， 3. 监听器， 4. 各种连接 5. 内部类和外部模块的引用 6. 单例模式    详情可见：http://note.youdao.com/noteshare?id=376631d4640128dc55646c8e577cc3ab

   而关于Java 如何处理它，我还不能给出准确答案，应该是使用各种检测泄漏的工具检测到之后，使用GC处理它们

* 垃圾收集器是什么?它是如何工作的
* 所有的对象实例都在JVM管理的堆区域分配内存
* 只要对象被引用，JVM就会认为它还存活于进程中。
* 一旦对象不再被引用，就不能被应用程序所访问，
* 垃圾收集器将删除它并重新声明未使用的内存。
* 比较 Arrays 和 ArrayLists。
* Arrays：一个包含许多和操纵数组有关方法的类，比如排序和查找，它继承自Object类。

   ArraysList：是一个容器，它可以实现数组的大小可变，方便地增加和删除元素。它实现了List接口的类。

   详细的笔记在：http://note.youdao.com/noteshare?id=659562b6beba34649d769d09cd2e0533

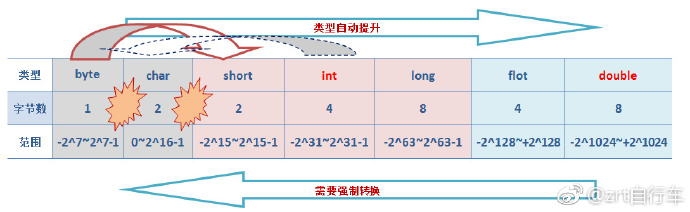
* 比较 HashSet 和 TreeSet。

   TreeSet是基于二叉树实现的，其中的数据是自动排序好的。不允许放入null值。

   HashSet是基于Hash表实现的，其中的数据是无序的，允许放入null值。

   详细的笔记在：http://note.youdao.com/noteshare?id=4a3e44e90105d9906eb308317bc816bb

* Java 中的类型转换。

   1. 基本数据的类型转换         在Java中，整数类型（byte/short/int/long）中，对于未声明数据类型的整形，其默认类型为int型。在浮点类型（float/double）中，对于未声明数据类型的浮点型，默认为double型。而当我们将一个数值范围较小的类型赋给一个数值范围较大的数值型变量，jvm在编译过程中会将此数值的类型进行自动提升。在数值类型的自动类型提升过程中，数值精度至少不应该降低（整型保持不变，float->double精度将变高）。而当我们需要将数值范围较大的数值类型赋给数值范围较小的数值类型变量时，我们需要手动地转换，称为强制类型转换。

   2.引用数据类型的类型转换

   由于继承和向上转型，子类向父类的转换是很自然的。但是当把父类转换为子类时，强制类型转换会在运行时检查父类的真实类型：如果引用的父类对象的真实身份是子类类型，只不过这个子类类型的对象经历了向上转型变成父类对象，这个时候我们再向下转回来子类，是可以的；而如果真的是父类的类型，就会抛出ClassCastException的异常。

   详细的笔记在：http://note.youdao.com/noteshare?id=7e85432e5d781ef8ad86ddabfc964885

* 方法重载和重写的区别
* 重载发生在编译时，重写发生在运行时。重载方法调用与其定义的绑定发生在编译时，但是重写方法调用与其定义的绑定在运行时发生。
* 静态方法可以重载，意味着一个类可以有多个同名的静态方法。静态方法不能被重写，即使在子类中声明了一个相同的静态方法，它与父类的相同方法无关。
* 最基本的区别是重载是在同一个类中完成的，重写父类需要有子类。重写是给父类的继承方法一个具体的实现。
* 静态绑定用于方法重载，动态绑定用于方法重写。性能：重载比重写更有效率，原因是方法重写的绑定是在运行时完成的。
* 私有方法和用 final 修饰的方法可以重载但不可重写。这意味着一个类可以有多个同名的 private/final 方法，子类不能重写父类的 private/final 方法。
* 方法重载的情况下不关心返回值类型， 它可以相同，也可以不同。但是方法重写的情况下可以有多个具体的返回值类型。  - 方法重载时参数列表必须不同，方法重写时参数列表必须相同。
* 什么是访问修饰符？它们能做什么？

诸如public，protected，private这几个关键词叫做访问修饰符。

其作用是控制它所定义的域或者方法的访问权。

* 接口可以继承另一个接口吗？

可以,通过继承，你可以

1. 在接口中添加新的方法声明
2. 还可以通过继承，在新的接口中组合数个接口，这时只需要用逗号将接口名一一分开来即可。

* Java 中 static 关键字是什么意思？

static是Java里的非访问修饰符，它可以用来创建类方法和类变量。

当修饰一个变量的时候，此变量就成了独立于对象的静态变量，无论一个类实例化了多少个对象，这个类只有一份这个静态变量的拷贝，所以static修饰的变量，即静态变量，也被叫做类变量。一个局部变量不能被声明为static变量。

当修饰一个方法的时候，此方法就成了独立于对象的静态方法，静态方法不能使用类的非静态变量，因为静态方法和静态变量先于非静态的其他成员初始化，静态方法先出来，然后才是非静态的，所以明白这个顺序很重要。静态方法从参数列表得到数据，然后计算这些数据。

有些人认为static方法不是面向对象的，因为它们确实具有全局函数的语义；使用static方法时，由于不存在this，所以不是通过「向对象发送消息」的方式来完成的。当代码中出现很多static方法时，你应该反思自己的设计的，然而，static确有其用。

总结一下，当static：

1. 修饰方法时，此方法能够不用在初始化对象的前提下直接调用，即，可以直接通过类名.static方法（）这样来访问。
2. 修饰代码块时，这部分代码块只会被执行一次。
3. 修饰变量时，这个变量只在内存中有一个副本。

* Java 中静态方法可以被重写吗？

严格来说，不存在静态方法的重写，当一个子类继承父类时，写同样的方法时，只是将父类的静态方法隐藏。

[静态方法能否被重写]（http://xm-king.iteye.com/blog/745787）

*所谓静态，就是在运行时，虚拟机已经认定此方法属于哪个类。 专业术语有严格的含义,用语要准确."重写"只能适用于实例方法.不能用于静态方法.对于静态方法,只能隐藏. 静态方法的调用不需要实例化.. 不实例化也就不能用多态了，也就没有所谓的父类引用指向子类实例.因为不能实例化 也就没有机会去指向子类的实例。所以也就不存在多态了。*

* 什么是多态？什么是继承？
* 多态是：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）

允许你将父对象设置成为一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作（摘自“Delphi4 编程技术内幕”）

* 继承是：子类继承父类的特征和行为，使得子类具有父类的各种属性和方法。
* Integer 和 int 之间的区别

1. Integer是int提供的封装类，而int是Java的基本数据类型； Integer默认值是null，而int默认值是0； 声明为Integer的变量需要实例化，而声明为int的变量不需要实例化； Integer是对象，用一个引用指向这个对象，而int是基本类型，直接存储数值
2. 两个new 出来的Integer总是不一样的。当使用==时，发现其内存地址不同，所以进行==返回false
3. 两个不是new出来的Integer，而是诸如Integer i = x，如果x的范围在-128～127，因为如下语句： Integer i5 = 127;//java在编译的时候,被翻译成-> Integer i5 = Integer.valueOf(127); 调用了Integer.valueOf (), 这会将127缓存，下次我们写：Integer i6 = 127;时，i6指向缓存中的同一个对象。所以此时i5==i6是true； 当x范围在-128～127之外，没有缓存存在，即使他们包裹的数值相等，他们也不能使用==的到true。
4. int和Integer的比较，无论Integer是否使用new，其值和将和平常的预料的一样:Integer自动拆箱，然后和int比较数值.这里就不会被内存地址的不同所影响，该相等时就相等。

* Java 中的对象是否会以引用传递或者值传递？详细说明。

 Java中的对象总是以值传递的。 ``` private void init(MyClass objVar) { objVar = new MyClass(); } public void test() { MyClass obj = null; init(obj); //在调用init方法后，obj仍旧指向空 //这是因为，obj是值传递，而不是引用传递。 }

```

* 什么是 ThreadPoolExecutor？ [Link](https://blog.mindorks.com/threadpoolexecutor-in-android-8e9d22330ee3)

 中文翻译请参考这里：http://note.youdao.com/noteshare?id=84c9bc5c8b01defcb9db9ca963bf6f67

* 本地变量、实例变量以及类变量之间的区别？

 本地变量就是局部变量，它在方法或者代码块里被声明并使用，其内存中的位置是栈里，没有默认初始化值，生命周期很短。   实例变量是没有被static修饰的成员变量，它属于一个类的一个实例。每次new一个实例，这样的变量也同时new一遍，其位置在堆区中，有默认初始化的值，生命周期和它所在的实例一样长。   类变量，又称静态变量，它是被static修饰的成员变量，它属于一个类，被所有实例共享。每次new一个实例，这样的变量并不会被new一遍，其内存位置在方法区内。可以通过类名直接访问。有默认的初始化值，生命周期很长。

* 什么是反射？ [Link](http://tutorials.jenkov.com/java-reflection/index.html)
* 在 Java 中什么是强引用、软引用、弱引用以及虚引用？

 强引用：不会被GC轻易清理，只要引用存在，垃圾回收器永远不会回收。

Object obj = new Object();

 软引用： 非必须引用，内存溢出之前进行回收 Object obj = new Object(); SoftReference<Object> sf = new SoftReference<Object>(obj); obj = null; sf.get();//有时候会返回null

 弱引用： 第二次垃圾回收时回收

 Object obj = new Object(); WeakReference<Object> wf = new WeakReference<Object>(obj); obj = null; wf.get();//有时候会返回null wf.isEnQueued();//返回是否被垃圾回收器标记为即将回收的垃圾

 虚引用： 垃圾回收时回收，无法通过引用取到对象值

 Object obj = new Object(); PhantomReference<Object> pf = new PhantomReference<Object>(obj); obj=null; pf.get();//永远返回null pf.isEnQueued();//返回是否从内存中已经删除

具体笔记可参考：http://note.youdao.com/noteshare?id=d52d8dd97e8dc162ddc90ff72a5c6001

 以及：http://note.youdao.com/noteshare?id=a9b2cebfbeb8fb33093ca318db58b389

* 什么是依赖注入？能说几个依赖注入的库么？你使用过哪些？
* 关键字synchronized的作用是什么？

synchronized通过修饰一个方法或者代码块，从而产生一个同步对象锁，以及对应 的同步代码块。每当有线程要对该同步代码块进行访问时，线程就会首先尝试获取这个对 象的锁，并在成功获取到对象锁后，对这个同步代码块进行正常的访问。在同步代码块的 访问过程中，线程会一直持有这个对象锁，直到同步代码块访问完毕，然后才会释放。

在上述线程持有同步锁并进行同步代码块访问过程中，其它线程将无法获得该对象锁，也 无法访问该同步代码，这些线程都会被阻塞直到上述线程访问完毕

* 为什么说String不可变的？

1. java.lang.String类型在实现时，其内部成员变量全部使用final来修饰，保证成员变量的引用值只能通过构造函数来修改；
2. java.lang.String类型在实现时，在外部某个地方，可能修改一个String实例的内部存储值的函数实现中，在这个地方的调用返回时，一律构造新的String对象或者新的byte数组或者char数组，给赋值符号的左边变量（左值）；

 详细请看：http://note.youdao.com/noteshare?id=747f5d269dd7e177153e4f099f095876

* 修饰符transient和volatile的作用是什么？

1. volatile

易式修饰符

出现它的原因是，java的多线程中存在两个或多个线程时间间隔很短地访问共享成员变量（指被多个线程共享的变量），在每个线程自己的工作内存中，可能对这个变量进行修改，但是没有及时将工作内存中的变量（对原本的共享成员变量的一份拷贝）写回共享成员变量，此时，当另外一个线程进行读取时，将无法得到最新的此变量，导致进程的工作不能正确进行。

于是出现了volatile，带有volatile修饰的变量，就是当其在某个线程自己的工作内存中发生改变时，会被强制地，写回公共成员变量所在的公共内存处。

如此便保证了，所有线程对这个变量的访问都是能得到此变量最新状态的访问。

1. transient

transient是一个类型修饰符，仅仅能用来修饰字段（变量）。在此字段所在的对象进行序列化的时候，这个字段不会被序列化。

其他没有transient修饰的变量将会被序列化，然后进行传输，或者存储到本地磁盘，transient变量就在这个过程里丢失了

* finalize()方法的作用是什么？

它是继承自Object类的方法，当垃圾回收器认为一个对象没有用的时候，会调用此对象的finalize（）方法，释放对象在堆中占用的内存。在finalize里，我们告诉了垃圾回收器应该做的事情。

* 异常捕获中的 try{}finally{} 块儿是如何工作的?

进行完try后，在try的return之前，总是会执行finally语句，进行善后处理。之后再进行return。

* 对象的实例化和初始化之间的区别是什么?

实例化可以理解为做出一个蛋糕：

Cake ca = new Cake();

初始化可以理解为，每个蛋糕都要有它自己的奶油：

ca.iceCream = "香草味奶油";

实例化着重于动作，强调==对象==从无到右的创建过程，而初始化着重于状态，强调==对象的特征==从无到有的过程。

* 静态块何时运行?

 初始化的顺序大致是这样的：

 1. 为类中的静态变量分配好空间（同时给变量以默认值） 2. 如果创建了类的对象，那么为类的实例变量分配地址空间（同时给变量以默认值）

就在这两步之间，发生了静态块的运行

* 解释一下 Java 中的泛型?
* StringBuffer 和StringBuilder 的区别在哪里?

 StringBuffer、StringBuilder和String一样，也用来代表字符串。String类是不可变类，任何对String的改变都 会引发新的String对象的生成；StringBuffer和StringBuilder则是可变类

 先说一下，以集合为例，HashTable是线程安全的，很多方法都是synchronized方法，而HashMap不是线程安全的，但其在单线程程序中的性能比HashTable要高。StringBuffer和StringBuilder类的区别也是如此，他们的原理和操作基本相同，区别在于StringBufferd支持并发操作，线性安全的，适 合多线程中使用。StringBuilder不支持并发操作，线性不安全的，不适合多线程中使用。新引入的StringBuilder类不是线程安全的，但其在单线程中的性能比StringBuffer高。

* StringBuilder 是怎么避免不可变字符串分配的问题？
* 什么是自动装箱和拆箱？

简单一点说，装箱就是 自动将基本数据类型转换为包装器类型；拆箱就是 自动将包装器类型转换为基本数据类型

自动装箱（autoboxing）:

一般来说，这是我们创建一个类的实例时，我们会这样：

```

MyClass a = new MyClass();

```

但是当我们新建一些支持自动装箱和拆箱的数据类型的时侯（比如Integer，基本数据类型的包装类），我们可以这样：

```

Integer i = 100;

```

执行这句语句，将会被编译器执行为：

```

Integer i = Integer.valueOf(100);

```

这就是自动装箱

自动拆箱（unboxing）:

将对象中的基本数据从对象中自动取出

比如：

```

Integer i = 10;//autoboxing int c = i;//unboxing

```

* 枚举和迭代器有什么区别？

1. 函数接口不同： Enumeration只有2个函数接口。通过Enumeration，我们只能读取集合的数据，而不能对数据进行修改。 Iterator只有3个函数接口。Iterator除了能读取集合的数据之外，也能数据进行删除操作。
2. Iterator支持fail-fast机制，而Enumeration不支持。 Enumeration 是JDK 1.0添加的接口。使用到它的函数包括Vector、Hashtable等类，这些类都是JDK 1.0中加入的，Enumeration存在的目的就是为它们提供遍历接口。Enumeration本身并没有支持同步，而在Vector、Hashtable实现Enumeration时，添加了同步。 而Iterator 是JDK 1.2才添加的接口，它也是为了HashMap、ArrayList等集合提供遍历接口。Iterator是支持fail-fast机制的：当多个线程对同一个集合的内容进行操作时，就可能会产生fail-fast事件。

* Java 中 *fail-fast* 和 *fail-safe* 的区别？
* 什么是 Java 优先级队列？
* 什么是设计模式 [Link](https://github.com/iluwatar/java-design-patterns)

### Android 核心

* 阐述一下 Activity 的生命周期。

创建 onCreate - 启动onStart – 开始 onResume – 暂停 onPause – 结束 onStop – 销毁onDestroy

每一个活动（ Activity ）都处于某一个状态，对于开发者来说，是无法控制其应用程序处于某一个状态的，这些均由系统来完成。 但是当一个活动的状态发生改变的时候，开发者可以通过调用 onXX() 的方法获取到相关的通知信息。

在实现 Activity 类的时候，通过覆盖（ override ）这些方法即可在你需要处理的时候来调用。

•onCreate ：当活动第一次启动的时候，触发该方法，可以在此时完成活动的初始化工作。 onCreate 方法有一个参数，该参数可以为空（ null ），也可以是之前调用 onSaveInstanceState（）方法保存的状态信息。

•onStart ：该方法的触发表示所属活动将被展现给用户。

•onResume ：当一个活动和用户发生交互的时候，触发该方法。

•onPause ：当一个正在前台运行的活动因为其他的活动需要前台运行而转入后台运行的时候，触发该方法。这时候需要将活动的状态持久化，比如正在编辑的数据库记录等。

•onStop ：当一个活动不再需要展示给用户的时候，触发该方法。如果内存紧张，系统会直接结束这个活动，而不会触发 onStop 方法。 所以保存状态信息是应该在onPause时做，而不是onStop时做。活动如果没有在前台运行，都将被停止或者Linux管理进程为了给新的活动预留足够的存储空间而随时结束这些活动。因此对于开发者来说，在设计应用程序的时候，必须时刻牢记这一原则。在一些情况下，onPause方法或许是活动触发的最后的方法，因此开发者需要在这个时候保存需要保存的信息。

•onRestart ：当处于停止状态的活动需要再次展现给用户的时候，触发该方法。

•onDestroy ：当活动销毁的时候，触发该方法。和 onStop 方法一样，如果内存紧张，系统会直接结束这个活动而不会触发该方法。

•onSaveInstanceState ：系统调用该方法，允许活动保存之前的状态，比如说在一串字符串中的光标所处的位置等。 通常情况下，开发者不需要重写覆盖该方法，在默认的实现中，已经提供了自动保存活动所涉及到的用户界面组件的所有状态信息。

* 谈谈 Android 的四大组件。

1. 活动：

Android 中，Activity是所有程序的根本，所有程序的流程都运行在Activity 之中，Activity可以算是开发者遇到的最频繁，也是Android 当中最基本的模块之一。在Android的程序当中，Activity 一般代表手机屏幕的一屏。如果把手机比作一个浏览器，那么Activity就相当于一个网页。在Activity 当中可以添加一些Button、Check box 等控件。可以看到Activity 概念和网页的概念相当类似。一般一个Android 应用是由多个Activity 组成的。这多个Activity 之间可以进行相互跳转，例如，按下一个Button按钮后，可能会跳转到其他的Activity。和网页跳转稍微有些不一样的是，Activity 之间的跳转有可能返回值，例如，从Activity A 跳转到Activity B，那么当Activity B 运行结束的时候，有可能会给Activity A 一个返回值。这样做在很多时候是相当方便的。 　　当打开一个新的屏幕时，之前一个屏幕会被置为暂停状态，并且压入历史堆栈中。用户可以通过回退操作返回到以前打开过的屏幕。可以选择性的移除一些没有必要保留的屏幕，因为Android会把每个应用的开始到当前的每个屏幕保存在堆栈中。

1. 服务

Service 是android 系统中的一种组件，它跟Activity 的级别差不多，但是他不能自己运行，只能后台运行，并且可以和其他组件进行交互。Service 是没有界面的长生命周期的代码。Service是一种程序，它可以运行很长时间，但是它却没有用户界面。这么说有点枯燥，来看个例子。打开一个音乐播放器的程序，这个时候若想上网了，那么，打开Android浏览器，这个时候虽然已经进入了浏览器这个程序，但是，歌曲播放并没有停止，而是在后台继续一首接着一首的播放。其实这个播放就是由播放音乐的Service进行控制。当然这个播放音乐的Service也可以停止，例如，当播放列表里边的歌曲都结束，或者用户按下了停止音乐播放的快捷键等。Service 可以在和多场合的应用中使用，比如播放多媒体的时候用户启动了其他Activity这个时候程序要在后台继续播放，比如检测SD 卡上文件的变化，再或者在后台记录地理信息位置的改变等等，总之服务嘛，总是藏在后头的。

开启Service有两种方式: （1） Context.startService（）：Service会经历onCreate -> onStart（如果Service还没有运行，则android先调用onCreate（）然后调用onStart（）； 　　　　如果Service已经运行，则只调用onStart（），所以一个Service的onStart方法可能会重复调用多次 ）； 　　　　StopService的时候直接onDestroy，如果是调用者自己直接退出而没有调用StopService的话，Service会一直在后台运行。该Service的调用者再启动起来后可以通过stopService关闭Service。 　　\*注意:多次调用Context.startservice（）不会嵌套（即使会有相应的onStart（）方法被调用），所以无论同一个服务被启动了多少次，一旦调用Context.stopService（）或者StopSelf（），他都会被停止。 　　补充说明：传递给StartService（0的Intent对象会传递给onStart（）方法。调用顺序为：onCreate --> onStart（可多次调用) --> onDestroy。 （2） Context.bindService（）：Service会经历onCreate（） -->onBind（），onBind将返回给客户端一个IBind接口实例，IBind允许客户端回调服务的方法，比如得到Service运行的状态或其他操作。这个时候把调用者（Context，例如Activity）会和Service绑定在一起，Context退出了，Srevice就会调用onUnbind --> onDestroyed相应退出，所谓绑定在一起就共存亡了。

1. 广播接收器

在Android 中，Broadcast是一种广泛运用的在应用程序之间传输信息的机制。而BroadcastReceiver 是对发送出来的Broadcast进行过滤接受并响应的一类组件。可以使用BroadcastReceiver 来让应用对一个外部的事件做出响应。这是非常有意思的，例如，当电话呼入这个外部事件到来的时候，可以利用BroadcastReceiver 进行处理。例如，当下载一个程序成功完成的时候，仍然可以利用BroadcastReceiver 进行处理。BroadcastReceiver不能生成UI，也就是说对于用户来说不是透明的，用户是看不到的。BroadcastReceiver通过NotificationManager 来通知用户这些事情发生了。BroadcastReceiver 既可以在AndroidManifest.xml 中注册，也可以在运行时的代码中使用Context.registerReceiver（)进行注册。只要是注册了，当事件来临的时候，即使程序没有启动，系统也在需要的时候启动程序。各种应用还可以通过使用Context.sendBroadcast （） 将它们自己的Intent Broadcasts广播给其他应用程序。

1. 内容提供者

Content Provider 是Android提供的第三方应用数据的访问方案。 　　在Android中，对数据的保护是很严密的，除了放在SD卡中的数据，一个应用所持有的数据库、文件等内容，都是不允许其他直接访问的。Andorid当然不会真的把每个应用都做成一座孤岛，它为所有应用都准备了一扇窗，这就是Content Provider。应用想对外提供的数据，可以通过派生Content Provider类， 封装成一枚Content Provider，每个Content Provider都用一个uri作为独立的标识，形如：content://com.xxxxx。所有东西看着像REST的样子，但实际上，它比REST 更为灵活。和REST类似，uri也可以有两种类型，一种是带id的，另一种是列表的，但实现者不需要按照这个模式来做，给id的uri也可以返回列表类型的数据，只要调用者明白，就无妨，不用苛求所谓的REST。

* Service 与 IntentService 的区别。[Link](https://stackoverflow.com/a/15772151/5153275)
* Android 应用的结构是什么？
* Android 应用中如何保存数据。
* 如何在 Android 应用中执行耗时操作。
* 两个 Fragment 之间如何通信。
* 阐述一下 Android 的通知系统。
* 两个不同的 app 之间如何交互。
* 什么是 Fragment？
* 为什么建议只使用默认的构造方法来创建 Fragment？[Link](https://stackoverflow.com/a/16042750/2809326)
* 为什么 Bundle 被用来传递数据，为什么不能使用简单的 Map 数据结构？
* 阐述一下 Fragment 的生命周期。[Link](https://www.techsfo.com/blog/wp-content/uploads/2014/08/complete_android_fragment_lifecycle.png)
* 如何理解 Android 的 Dialog ？
* 解释下 Android 的 View 。
* 你能创建自定义 View 吗？具体是如何创建的？
* 什么是 ViewGroup ，它与 View 的区别在哪里？
* Fragment 和 Activity 有什么区别？它们之间又有什么关系？
* 谈谈 Serializable 接口和 Parcelable 接口的区别。在 Android 中最好使用哪种接口？
* Activity 的启动模式有哪些？[Link](https://blog.mindorks.com/android-activity-launchmode-explained-cbc6cf996802)
* 解释一下 Android 中的 Intent 。[Link](https://stackoverflow.com/questions/6578051/what-is-an-intent-in-android)
* 什么是隐式 Intent ？
* 什么是显式 Intent ？
* 解释一下 AsyncTask 。
* 如何理解 Android 中的广播。[Link](https://stackoverflow.com/questions/5296987/what-is-broadcastreceiver-and-when-we-use-it)
* 如何理解 Android 的 LocalBroadcastManager 。[Link](https://developer.android.com/reference/android/support/v4/content/LocalBroadcastManager.html)
* 什么是 JobScheduler ？[Link](http://www.vogella.com/tutorials/AndroidTaskScheduling/article.html)
* 什么是 DDMS ？你可以用它来做什么？
* 解释一下什么是 support libary ，以及为什么要引入 support library ？[Link](http://martiancraft.com/blog/2015/06/android-support-library/)
* 如何理解 Android 中的 ContentProvider 。它通常用来干什么？
* 什么是 Data Binding ？[Link](https://developer.android.com/topic/libraries/data-binding/index.html)
* Android 的核心组件具体都有什么？[Link](https://developer.android.com/topic/libraries/architecture/index.html)
* 什么是 ADB ？
* 什么是 ANR ？如何避免发生 ANR ？
* AndroidManifest.xml 是什么？
* 解释一下 broadcast 和 intent 在 app 内传递消息的工作流程。
* 当 Bitmap 占用较多内存时，你是怎么处理的？
* Android 应用有哪些不同的存储数据的方式？
* 什么是 Dalvik 虚拟机？
* AsyncTask 的生命周期和(它所属的) Activity 的生命周期有什么关系？这种关系可能会导致什么样的问题？ 如何避免这些问题发生？
* Intent filter 是用来做什么的？
* 什么是 Sticky Intent？[Link](http://www.androidinterview.com/what-is-a-sticky-intent/)
* 什么是 AIDL ？列举一下通过 AIDL 创建被绑定的服务（bounded service）的步骤。
* Android 的权限有多少个不同的保护等级？
* 在转屏时你如何保存 Activity 的状态？[Link](https://stackoverflow.com/questions/3915952/how-to-save-state-during-orientation-change-in-android-if-the-state-is-made-of-m)
* 相对布局和线性布局的区别。
* 如何实现 XML 命名空间？
* View.GONE 和 View.INVISIBLE 之间的区别。
* Bitmap 和 .9（nine-patch）图片之间有什么区别？
* 谈谈位图池。[Link](https://blog.mindorks.com/how-to-use-bitmap-pool-in-android-56c71a55533c)
* 在 Android 中如何避免内存泄漏？
* Android 桌面的小部件是什么？
* 什么是 AAPT ？
* 你是如何在 Android 应用程序中发现内存泄漏的？
* 你如何排查应用崩溃的原因？
* 为什么你应该避免在主线程上运行非用户界面相关的代码？
* 你是如何适配不同分辨率的手机的？
* 如何理解 Doze 模式。如何理解应用程序待机模式（App Standby）。
* 在 Android 中，你可以使用什么来进行后台操作?
* 什么是 ORM ？它是如何工作的？
* 什么是 Loader ？
* 什么是 NDK ，为什么它是有用的？
* 如何理解严格模式（StrictMode）。 [Link](https://blog.mindorks.com/use-strictmode-to-find-things-you-did-by-accident-in-android-development-4cf0e7c8d997)
* 什么是 Lint ？它的用途是什么？
* 什么是 SurfaceView ？
* ListView 和 RecyclerView 有什么区别？
* 什么是 ViewHolder 模式？为什么我们应该使用它？
* 什么是 PendingIntent ？
* 你能手动调用垃圾回收吗？
* 周期地更新页面的最好方式是什么？
* 有哪些类型的广播？
* 你开发过组件吗？请描述一下。[Link](https://blog.mindorks.com/android-widgets-ad3d166458d3)
* 如何理解上下文（Context）。怎么使用它？[Link](https://medium.com/p/understanding-context-in-android-application-330913e32514)
* 你知道什么是视图树(View Tree)吗？怎样优化它的深度？
* onTrimMemory() 方法是什么？
* Android 应用可以使用多进程吗？怎样使用？
* 内存溢出（OutOfMemory）是怎么发生的？
* 文本样式接口（Spannable）是什么？
* 什么是过度绘制（overdraw）？
* 什么是渲染脚本（renderscript）？[Link](https://blog.mindorks.com/comparing-android-ndk-and-renderscript-1a718c01f6fe)
* Dalvik 虚拟机模式和 ART（Android Runtime）虚拟机模式的区别。
* FlatBuffers 和 JSON 的区别。[Link](https://blog.mindorks.com/why-consider-flatbuffer-over-json-2e4aa8d4ed07)
* 谈谈 Android 的注解。[Link1](https://blog.mindorks.com/creating-custom-annotations-in-android-a855c5b43ed9), [Link2](https://blog.mindorks.com/improve-your-android-coding-through-annotations-26b3273c137a)
* 描述一下约束布局（Constraint Layout）。[Link](https://blog.mindorks.com/using-constraint-layout-in-android-531e68019cd)
* 阐述一下 Android 中的 HashMap , ArrayMap 和 SparseArray 。[Link](https://blog.mindorks.com/android-app-optimization-using-arraymap-and-sparsearray-f2b4e2e3dc47)
* 阐述一下 Looper, Handler 和 HandlerThread 。[Link](https://blog.mindorks.com/android-core-looper-handler-and-handlerthread-bd54d69fe91a)
* 如何降低 Android 应用的耗电量？[Link](https://blog.mindorks.com/battery-optimization-for-android-apps-f4ef6170ff70)
* SnapHelper 是什么？[Link](https://blog.mindorks.com/using-snaphelper-in-recyclerview-fc616b6833e8)
* 在 Android 中怎么处理多点触控？[link](https://arjun-sna.github.io/android/2016/07/20/multi-touch-android/)

### 架构

* 请介绍一下你做的上一个 App 的架构。
* 请介绍一下 MVP。 [Link](https://blog.mindorks.com/essential-guide-for-designing-your-android-app-architecture-mvp-part-1-74efaf1cda40)
* Presenter 是什么？
* 什么是模型？
* 请介绍一下 MVC。
* Controller 是什么？
* 请介绍一下 MVVM。 [Link](https://github.com/MindorksOpenSource/android-mvvm-architecture)
* 谈谈你对代码整洁之道（clean code）的理解。[Link](https://blog.mindorks.com/every-programmer-should-read-this-book-6755dedec78d)

### 设计问题

* 请设计 Uber App。
* 请设计 Facebook App。
* 请设计 Facebook Near-By Friends App。
* 请设计 WhatsApp。
* 请设计 SnapChat。
* 基于地理位置 App 的设计问题。

### 工具和技能

* Git. [Link](https://github.com/git-tips/tips)
* RxJava. [Link](https://blog.mindorks.com/a-complete-guide-to-learn-rxjava-b55c0cea3631)
* Dagger 2. [Link](https://medium.com/p/a-complete-guide-to-learn-dagger-2-b4c7a570d99c)
* Android 开发实用工具。 [Link](https://blog.mindorks.com/android-development-useful-tools-fd73283e82e3)
* Firebase. [Link](https://firebase.google.com/)

### Android 测试驱动开发

* Espresso 是什么？ [Link](https://developer.android.com/training/testing/ui-testing/espresso-testing.html)
* Robolectric 是什么？ [Link](http://robolectric.org/)
* UI-Automator 是什么？ [Link](https://developer.android.com/training/testing/ui-testing/uiautomator-testing.html)
* 请解释一下单元测试。
* 请解释一下设备化测试。
* 你是否做过单元测试或者自动测试？
* 为什么要使用 Mockito？ [Link](http://site.mockito.org/)
* 请描述一下 JUnit 测试。

### 其他

* 描述一下 REST APIs 如何工作 ？
* 描述一下 SQLite 。
* 描述一下 数据库 。
* 项目管理工具 - trello ，basecamp ，kanban ，jira ，asana 。
* 关于构建系统 - gradle , ant , buck 。
* APK 逆向工程 。
* 混淆器用于什么 ？
* 什么是混淆？ 用于什么？ 如何压缩 ？
* 如何构建你的发布版本的 APP ？
* 如何面向特定用户群体更新应用程序版本 ？
* 可以识别卸载我们的应用程序的用户吗 ？
* 缩小 APK 的体积 。[Link](https://blog.mindorks.com/how-to-reduce-apk-size-in-android-2f3713d2d662)
* Android 开发最佳实践 。[Link](https://blog.mindorks.com/android-development-best-practices-83c94b027fd3)
* Android 代码风格和指南 。[Link](https://blog.mindorks.com/android-code-style-and-guidelines-d5f80453d5c7)
* 你尝试使用过 Kotlin 吗 ？[Link](https://blog.mindorks.com/why-you-must-try-kotlin-for-android-development-e14d00c8084b)
* 开发 Android 应用程序时应该连续测量哪些指标 ？[Link](https://blog.mindorks.com/android-app-performance-metrics-a1176334186e)

### 贡献者

感谢这些无私的贡献者，排名不分先后。

[mengxn](https://github.com/mengxn)、[innovatorCL](https://github.com/innovatorCL)、[SmartNJ](https://github.com/SmartNJ)、[Zhiw](https://github.com/Zhiw)、[lanyuanxiaoyao](https://github.com/lanyuanxiaoyao)、[934079371](https://github.com/934079371)、[cdevelopr](https://github.com/cdevelopr)、[smartbeng](https://github.com/smartbeng)、[ikook](https://github.com/china-kook)、[mrfanr](https://github.com/mrfanr)、[androidZzT](https://github.com/androidZzT)、[qiaojialin](https://github.com/qiaojialin)、[maokai1229](https://github.com/maokai1229)、[renxuelong](https://github.com/renxuelong)、[dzx1994](https://github.com/dzx1994)

### License

Copyright (C) 2017 stormzhang  
  
 Licensed under the Apache License, Version 2.0 (the "License");  
 you may not use this file except in compliance with the License.  
 You may obtain a copy of the License at  
  
 http://www.apache.org/licenses/LICENSE-2.0  
  
 Unless required by applicable law or agreed to in writing, software  
 distributed under the License is distributed on an "AS IS" BASIS,  
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
 See the License for the specific language governing permissions and  
 limitations under the License.