#### 提纲

#### （一） java基础面试知识点

#### java中==和equals和hashCode的区别（基本数据类型和引用类型）

<https://baijiahao.baidu.com/s?id=1652442464525126879&wfr=spider&for=pc>

对于基本数据来说，==比较的是数值是否相等

如果引用类型没有重写了equals，那么equals和==比较的都是引用的地址是否相同，

如果引用类型重写了equals方法，那么equals比较的就是内容是否相等，比如String.

==比较的则是引用的内存地址是否相等

如果重写了equals方法，就必须重写hashcode方法，因为equals默认比较的是对象的引用地址，而想要比较两个对象的内容是否相等，就必须保证这两个对象的hashcode是一致的，重写hashcode的方法：result = 31 \* result + filed.hashcode

#### int、char、long各占多少字节数（4,2,8）

#### int与integer的区别

#### 探探对java多态的理解（多态是什么？作用？体现在哪？实现方式？机制？代码）

同一个接口，不同子类实现后变现出不一样的结果就是多态。

多态能降低耦合度，具有更好的扩展性，灵活性和简化性。

具体体现在继承、重写和向上或者向下转型这几个主要的方面。

实现：子类继承父类，重写父类方法；子类实现接口；子类继承抽象类并实现抽象方法；

机制：java在程序运行的过程中才决定让引用变量绑定到哪个实例化对象上，所以可以支持多态。

向上/向下转型的基础是具有继承关系。向上转型：子类将失去自身的方法

向下转型：需要强制转换。

向上转型和向下转型的意义：在进行泛型编程的时候，向下转型就很有用。

比如

<https://blog.csdn.net/xyh269/article/details/52231944>

#### String、StringBuffer、StringBuilder区别

String会创建一个新的对象指向改变的字符串，一旦对象创建完成，这个字符串就不可再更改；而StringBuilder和StringBuffer对象能被多次修改而不会创建新的对象；前者不是线程安全的，所以速度也更快。

#### 什么是内部类？内部类的作用

定义在一个类内部的类叫做内部类。分为成员内部类、局部内部类、匿名内部类和静态内部类。

成员内部类默认是持有外部类的一个引用的，静态内部类与外部类不关。

作用：内部类可以单独继承父类或者实现接口，不受外部类的限制，所以从一定程度上来说是对java单继承的弥补；同时可以对外隐藏，更方便写事件驱动程序等。

#### 抽象类和接口区别（从变量、方法、使用比较）

1：抽象类的成员变量可以是任何类型，接口只能是public static finnal

2：抽象类的方法可以是抽象方法也可以是普通方法，接口只能定义抽象方法。（jdk 1.8后接口也可以有静态方法，但是必须要有方法体）

3：抽象类可以有方法体，接口只能定义方法声明

4：一个类可以继承一个抽象类，但是可以实现多个接口

#### 抽象类与接口的应用场景

抽象类是一种对事物属性和构成的抽象，接口是对一种行为或者目的进行抽象。使用场景看业务和功能定，比如存在多个子类具有一些相同的属性，可以进行复用的，就可以定义抽象类；不同的子类需要实现不同的行为效果，那就使用接口。

#### 抽象类是否可以没有方法和属性？（可以）

#### 泛型中<? extends X>和<? super X>的区别

*/\*  
泛型上界通配符  
 \* 道理很简单，？ extends X 表示类型的上界，类型参数是X的子类，那么可以肯定的说，  
 \* get方法返回的一定是个X（不管是X或者X的子类）编译器是可以确定知道的。  
 \* 但是set方法只知道传入的是个X，至于具体是X的那个子类，不知道，就是向下转型可以但是向上转型不确定。  
 \* 总结：主要用于安全地访问数据，可以访问X及其子类型，并且不能写入非null的数据。  
 \*/*

*/\*  
泛型下界通配符  
 \* ？ super X 表示类型的下界，类型参数是X的超类（包括X本身），  
 \* 那么可以肯定的说，get方法返回的一定是个X的超类，那么到底是哪个超类？不知道，  
 \* 但是可以肯定的说，Object一定是它的超类，所以get方法返回Object。  
 \* 编译器是可以确定知道的。对于set方法来说，编译器不知道它需要的确切类型，但是X和X的子类可以安全的转型为X，  
 就是向下转型一定可以转为X，但是向上转型不知道确定要转为哪一个。  
 \* 总结：主要用于安全地写入数据，可以写入X及其子类型。  
 \*/*

#### 父类的静态方法能否被子类重写

不能。静态方法在类加载的时候已经绑定了，指向了一个固定的地址，只跟类有关，跟实例对象无关，所以不能被重写。如果子类有一个跟父类静态方法重名的方法，那方法也只是一个新的方法而已，不是重写。

#### 进程和线程的区别（进程是啥？线程是啥？两者的关系）

<https://www.cnblogs.com/aflyun/p/10575740.html> JVM内存模型

1：进程是资源分配的最小单位，线程是程序执行的最小单位。

2：进程有自己独立的地址空间，线程只是共享进程的数据，使用的是相同的地址空间。也就是一种包含的关系，一个进程包含有多个线程。

堆：是大家公有的空间，在系统对进程初始化的时候分配。

栈：是线程独有的，在线程初始化的时候创建。

方法区、堆、虚拟机栈、本地方法栈、程序计数器

#### final，finally，finalize的区别

Final修饰类的时候表示该类不允许被继承、修饰方法的时候表示不允许子类对该方法进行重写、修饰变量表示该变量不可更改。

Finally是异常处理的一部分，在try语句执行了的情况下，finally最终会执行，可用于释放资源操作。

Finalize（）是Object的一个方法，在必要的时候程序员可以使用它来启动GC操作。

#### Serializable 和Parcelable 的区别

使用场景：在进行Android开发的时候，没办法将对象的引用传递给Activity或者fragment,需要将对象放到Intent或者Bundle里面，然后再进行传递。

一：实现Serializable接口。注意这种方式不能序列化静态成员变量和transient修饰的变量。注意，当进行反序列的时候，JVM会把字节流的seriaVersionUID与本地实体类的seriaVersionUID进行比较，如果不一样的会造成反序列化失败，所以需要手动生成系列号。利用了反射，需要频繁GC所以更慢。

二：实现Parcelable接口。Android独有，效率更高。作用是可以永久性保存对象、可以把序列化对象在网络中传播、可以在进程间传递对象。Activity可以通过putExtra()传入序列化好的对象，通过getParcelableExtra()获取对象。在内存中操作，所以更快。

两者区别：第一种方式能很好地将数据保存在磁盘上，但是会引起频繁的GC；第二种方式效率高。

**静态属性和静态方法是否可以被继承？（可以，隐藏）是否可以被重写（不可以）？**以及原因？

#### 成员内部类、静态内部类、局部内部类和匿名内部类的理解，以及项目中的应用

<https://www.cnblogs.com/dolphin0520/p/3811445.html>

#### 谈谈对kotlin的理解(更加简洁，代码更少)

1：相对于java，代码更加简洁。

2：在编译的时候就能检查到是否有空指针，如果有，编译不通过。

3：跟java可以互相调用，还能混合编程。

#### string 转换成 integer的方式及原理（逐一用字串的Assica码减去‘0’字符的assica码值）

##### （二） java深入源码级的面试题（有难度）

**哪些情况下的对象会被垃圾回收机制处理掉？**（对象无引用，可达性分析，常量和静态属性可作为节点）

可达性分析：从一个根节点一直往下搜索，所有走过的路径就会形成一条GC Roots,如果一个对象到GC Roots没有任何引用相连，那就是不可用对象。因为引用都是存储在栈中，所以可以作为GC Roots的结点有虚拟栈中对象的引用、本地方法栈中JNI对象的引用、方法区中静态引用的对象

引用计数法、标记-清除算法、标记-整理算法、分代回收算法

Full GC的原因

1：老年代被写满

2：显式调用System.gc()

要尽可能避免Full GC

**静态代理和动态代理的区别，什么场景使用？（**太难了）

**Java的异常体系（**运行时异常等）

运行时异常不需要程序员捕获，比如空指针异常、数组越界异常、强制转换异常、数据格式异常等；编译时异常需要程序员做异常处理，否则编译不通过，比如FileNotFound、IOExcepetion、SqlExcepation。

#### 修改对象A的equals方法的签名，那么使用HashMap存放这个对象实例的时候，会调用哪个equals方法？（使用该对象的equals方法）

**Java中实现多态的机制是什么？**（运行时绑定）

**说说你对Java反射的理解**（看项目代码）

在运行状态中，对于任何一个类都能够获取到它的方法和属性；对于任何一个对象，都能够调用它的方法和属性，这种动态调用方法的特性就叫做反射。

获取Class对象的方法：Class cls = 实例对象.getClass();或者Class cls = 类.class;或者

Class cls=Class.forName(“xxx.xxx.xxx”);

#### 说说你对Java注解的理解 ？

**说一下泛型原理，并举例说明**（看代码实现）、？

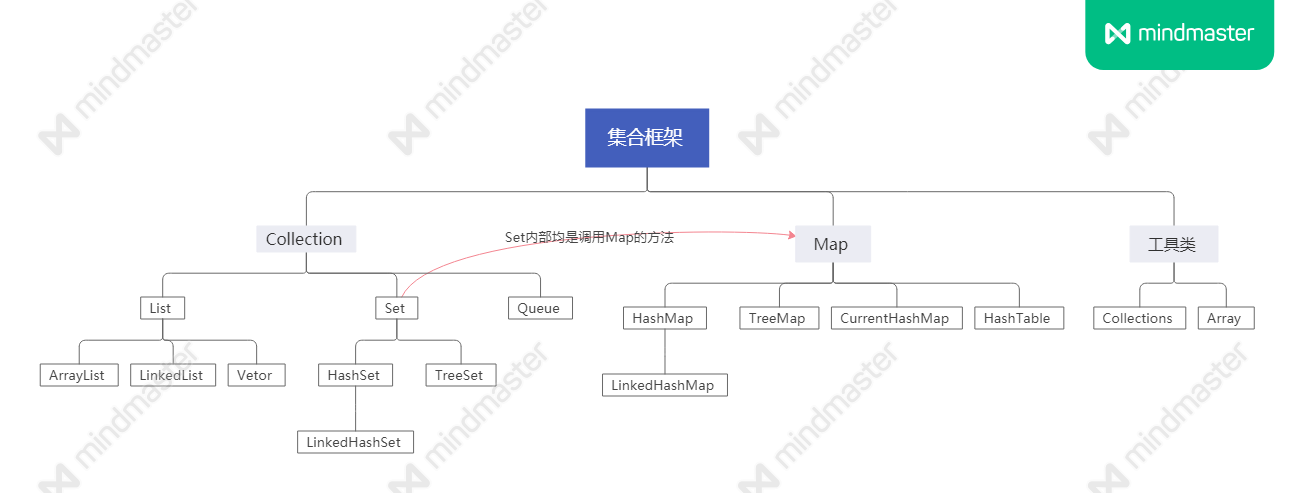
**Java中String的了解**（不可变，存储在堆中的常量池）

**String为什么要设计成不可变的？（**安全、减少资源）

##### 数据结构

大佬博客地址：<https://www.cnblogs.com/LiaHon/>

**列举java的集合以及集合之间的继承关系**（看思维导图）



**List,Set,Map的区别**

List存储存储一组不唯一且有序的元素。

Set存储一组唯一且无序的元素。

Map存储一组键值对象，提供key到value的映射。

**List和Map的实现方式以及存储方式**

List通过索引来访问元素，以数组的形式存储。

Map是通过数组+链表实现的，以键值对的形式存储。

**HashMap的实现原理（散列表：数组+链表）**

Hashmap底层采用了数组+链表的数据结构。记录了键值对的数量，同时有一个默认的加载因子0.75，用于衡量满的程度，当键值对的数量达到容量临界值的时候，就会进行扩容，扩容规则是2的幂次。（hashMap在jdk 1.8之后才引入红黑树，之前是没有的）

**HashMap如何put数据（从HashMap源码角度讲解，存储过程）？**

1：通过hash算法得到hash值，即存储位置。

2：判断table是否是nul或者长度为0，如果是的话就进行扩容。

3：判断table中hash对应的位置是否已经有了元素。（即桶的位置）

4：如果table[i]没有元素，那么就新建结点并插入，随后判断键值对的数量是否达到了临界值，如果达到了就进行扩容。

5：如果table[i]有元素，就判断要传入的key是否已经存在了，存在的话就直接覆盖value。如果value也相同的话就直接返回旧值，不再存储。这就是判断元素重复的方法。

6：如果要传入的key不存在，接着判断table[i]是否是treeNode，如果是就用红黑树插入键值对；

7：如果不是treeNode，就判断链表的长度是否大于等于8了，不大于的话就连边插入键值对。

8：链表长度大于等于8的话，就转化为红黑树再插入。

针对第一步：为什么不直接使用key.hashcode()来获取hash值？

因为hash的目的是为了得到索引，如果hash过于简单，就会有很大的几率产生hash冲突，通过混合高位低位得出来的hash值，随机性更大，冲突的概率就小。

**HashMap又是如何get数据的？**

1：传入的key计算hash值找到数组的下标；

2：如果该下标对应的节点的key值刚好一样就返回value;

3：否则就寻找下一个节点，判断是否是treeNode,如果是就遍历红黑树；

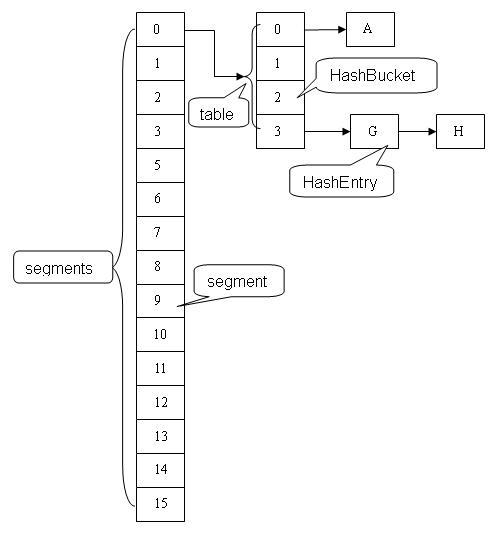
4：如果不是treeNode，就遍历链表。

**HashMap的缺点：非线程同步的，并发时会出问题。ConcurrentHashMap就是为了解决这个问题诞生的。**

**ConcurrentHashMap的实现原理**

<https://blog.csdn.net/weixin_44460333/article/details/86770169>

Jdk 1.8以前。ConsurrentHashMap采用锁分段技术，将数据分成一段一段的，给每一段数据分配一把锁，当一个线程占用其中一个数据段的时候不影响其他线程对其他数据段的访问和修改。它的结构如下图

****

JDK 1.8之后已经放弃了Segment分段锁机制。采用**CAS+Synchronized**来保证并发的安全。数组+链表+红黑树的结构。注意点：内部原理和hashMap类似，不同的是当table需要扩容的时候，concurretnHashMap会先创建一个table,把旧的table的内容复制过去。用了volatile修饰，保证了线程之间的可见性。

Tip:什么是CAS,即只有当变量的预期值和实际值相同的时候才把新值更新进内存。属于乐观锁。让线程不停的去尝试更新，自旋。

<https://blog.csdn.net/wengyupeng/article/details/90239411>

**ArrayMap和HashMap的对比（**ArrayMap是android API）

ArrayMap的设计考虑了内存的优化。采用两个数组进行数据的存储，一个数组记录key的hash值，一个数组记录value的值。Key可以为任何类型。

SparseArray同样是使用了两个数组进行数据存储，在存储和查找数据的时候都使用了二分法进行查找，但是它的key只能是整型的。

使用场景的对比：当数据量比较大的时候使用HashMap,反之使用ArrayMap,同时又可以根据key的类型决定使用ArrrayMap还是SparseArray。

**HashTable实现原理**（Synchronized整段加锁+hashMap结构）

<https://www.cnblogs.com/baojun/p/11086989.html>

1：HashMap允许key和value都为null,但是HashTable不允许，当出现null的时候，HashTable会抛出空指针异常。

2：HashTable底层方法大量使用了Synchronized修饰，所以可以实现线程同步。

（创建线程同步的HashMap方法：Collections中有一个静态方法synchronizedMap会返回一个线程安全的map对象）

**TreeMap具体实现（**红黑树结构，支持key值排序）

treeMap通过红黑树实现的。默认情况下支持按照key值的自然顺序进行排序；可被克隆，可序列化。

Put数据的过程：

1：判断红黑树是否已经创建，如果没有创建，就new Entry创建红黑树并赋值到root结点

2：如果红黑树已经建立，遍历结点的key值和传入的值是否相等，找到相等的key就更新value

3：找不到相等的key就遍历到最后的结点，使之作为待插入的结点的父节点。

4：插入子节点并调整红黑树使得满足规则。

Get数据的过程：

1：利用二分查找从根节点开始找，比较传入的key和根节点的key

2：传入的key比当前结点的key小，就到当前结点的左子树中找

3：传入的key比当前结点的key大，就到当前结点的右子树中找

4：相等的话就直接返回value值。如果循环，直到最后一个结点。

**HashMap和HashTable的区别**

1：继承自不同的父类

2：hashMap可以有一个key为null,多个value为null;HashTable不允许有null

3：前者不是线程安全的，后者是线程安全的。

4：Hash值的计算方式不一样

**LinkHashMap的实现原理**

<https://www.cnblogs.com/LiaHon/p/11180869.html>

底层采用数组+双向链表/红黑树实现，可以在HashMap的基础上实现顺序读取，即先put进去的会先get出来，顺序保持不变。

Put数据的过程：

\* 将新创建的节点p作为尾结点tail，

\* 当然如果存储的第一个节点，那么它即是head节点，也是tail节点，此时节点p的before和after都为null

\* 否则，建立与上一次尾结点的链表关系，将当前尾节点p的前一个节点（before）设置为上一次的尾结点last,

\* 将上一次尾节点last的后一个节点（after）设置为当前尾结点p

\* 通过此方法实现了双向链表功能，完成before,after,head,tail的值设置

Get过程：

在hashMap的get的基础上，将最后传入的结点移动到最后。

**HashMap与HashSet的区别**（HashSet实现的是Collection接口，内部还是HashMap实现）

hashSet的底层都是通过HashMap实现的。HashSet添加的是集合，它把要添加的元素放在HashMap的key的位置上，而value用一个默认值代替。

**HashSet与HashMap怎么判断集合元素重复？**（根据hashcode和equals方法进行判断）

如果传入的元素和当前元素的Hash值相等并且key值相同或者equals返回true,那么就认为元素存在了，不再进行添加。

**LinkHashSet、TreeSet和LinkHashMap、TreeMap的区别**（Set的内部都是基于Map实现的）

<https://www.cnblogs.com/LiaHon/p/11249560.html>

**集合Set实现Hash怎么防止碰撞**（重写hashcode和equals方法）

**解决hash冲突的办法都有些什么？**

<https://blog.csdn.net/seulzz/article/details/77163878>

1：开放地址法。即一旦发生冲突，就去寻找下一个空的散列地址，并插入数据。

2：再hash法。发生冲突后再次用其他hash函数计算hash值

3：拉链法。发生冲突的时候，把冲突的结点通过next指针连接起来。HashMap就是使用的这种办法。

**ArrayList实现原理（数组）**

底层是一个Object数组，当调用new ArrayList<>()的时候，集合的长度为0；也可以传一个初始长度进去；调用new ArrayList<>(new HashSet())的时候，内部会将其转为数组；可以传递任何实现了Collection接口的类进去。

ArrayList的扩容机制：数组当前足够的最小容量\*1.5，比如当前元素有10个，但是数组只有8，那么就会扩容为 10\*1.5 = 15

Set过程：通过传入的下标找到当前位置，判断是否越界，替换对应位置的值。

Remove过程：

1：检查下标是否越界

2：通过下标找到待删除的元素并返回

3：计算待删除元素需要移动的位数size-index-1

4：如果待删除的元素是最后一位，就直接设置为null

5：如果待删除的元素不是最后一位，就把除掉该元素以外的数组复制到一个新的数组上，并将待删除元素设置为null,通过设置null触发GC清理

**LinkedList实现原理**

<https://www.cnblogs.com/LiaHon/p/11107245.html>

底层采用双向链表实现。

**get**数据的实现机制：比较传入的索引和集合长度size/2做比较，如果index小，就从第一个顺序查找；如果index大，就从最后一个倒序查找。（这就是for循环慢的原因）

其实LinkedList的增删一样要遍历找到对应的元素，增删效率比ArrayList高的原因是因为**省去了ArrayList插入数据的时候可能需要扩容或者移动元素或者复制元素到新数组造成的开销**，但是扩容、移动、复制元素并不一定发生，所以LinkedList的高效是相对的。

遍历效率：LinkedList使用迭代器遍历的效率远远高于普通for循环。

**ArrayList和LinkedList和Vetor的区别，以及应用场景（**前者数组，双向链表，数组）

<https://www.cnblogs.com/LiaHon/p/11109104.html>

| **类型** | **线程安全** | **内部结构** |  | **扩容规则** | **执行效率** | **序列化** |
| --- | --- | --- | --- | --- | --- | --- |
| ArrayList | 否 | 数组Object[] | 10 | 数组足够最小长度\*1.5 | 高 | 是 |
| Vector | 是 | 数组Object[] | 10 | 默认数组足够最小长度\*2，可自定义每次扩容数量 | 低 | 是 |

| **类型** | **内部结构** | **插入效率(正常情况)** | **删除效率(正常情况)** | **顺序遍历效率** | **随机遍历效率** | **占用内存** | **序列化** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ArrayList | 数组Object[] | 低 | 低 | 高 | 高 | 低 | 是 |
| LinkedList | 双向链表Node | 高 | 高 | 高 | 低 | 高 | 是 |

ArrayList和LinkedList的效率并不是绝对的，看具体场景。

**二叉树的深度优先遍历和广度优先遍历的具体实现**（深度遍历用栈，广度遍历用队列）

**堆的结构（**大根堆，小根堆）

**堆和树的区别**

**什么是深拷贝和浅拷贝**

浅拷贝：会创建一个新的对象，有着原始对象属性值的一份拷贝，如果是基本类型，那拷贝的是数值；如果是引用类型，拷贝的就是内存地址，因此如果一个对象改变了这个地址，就会影响到另一个对象；

浅拷贝实现方式:实现Cloneable接口，并覆写clone()方法，通过super.clone。

深拷贝：会创建新对象，对于基本数据类型会拷贝数值；如果是引用类型，会开辟一段新的内存空间，然后拷贝内容；对于多层对象嵌套的，每个对象都需要实现Cloneable对象并重写clone（）方法。速度慢，内存花销比较大。

深拷贝实现方法:每一层实现浅拷贝，最后再顶层的类中的clone方法中调用所有类的clone()方法即可实现深拷贝。

**讲一下对树，B+树的理解**

**开启线程的三种方式？**

继承Thread、实现Runable、匿名内部类实现Runable

**run()和start()方法区别**

start启动线程，run只是调用方法

**如何控制某个方法允许并发访问线程的个数**？（Samaphore,代码）

**在Java中wait和seelp方法的不同；**（wait属于Object,释放锁资源；sleep属于线程，不释放锁）

**谈谈wait/notify/notifyAll关键字的理解**（看代码）

Wait暂停一个线程，notify随机唤醒一个线程

调用wait和notify的必须是同一个对象。调用前需要先取得该对象的锁资源，可以使用synchronized来获取对象的控制权。

在哪个线程里面调用wait方法，挂起的就是哪个wait方法。

使用者两个方法可以实现简单的生产者消费者模型。

**什么导致线程阻塞？**（sleep,wait，执行同步代码，无法获得锁；执行IO操作，等待资源）

**线程如何关闭？（**标记位退出、interrupt()、stop()弃用，看代码）

**讲一下java中的同步的方法（synchronized**、volatile、ReentrantLock、ThreadLocal）

**如何保证线程安全？**（原子性、可见性、有序性）

线程安全主要体现在

1原子性：提供互斥访问，同一个时刻只能有一个线程对数据进行操作。（atomic,synchronized，ReentrantLock）

2可见性：一个线程对主存的修改可以及时的被其他线程看到。（synchronize，volatile）

3有序性：执行执行顺序。

**线程间操作List（**分段操作，每个线程处理一段数据）

**synchronize的原理**

依赖JVM实现。

**Synchronized用法**（修饰代码块、成员方法、类、静态方法）

<https://www.cnblogs.com/lukelook/p/9946065.html>

当两个并发线程访问同一个对象的synchronized代码块的时候，在同一个时刻只能有一个线程得到执行，两一个线程会被阻塞，直到当前线程执行完毕才能执行。换言之，多个线程访问不同的对象，是不会造成阻塞的，因为每一个对象都有一个锁。

一个线程访问一个对象的synchronized代码块的时候，别的线程一样可以访问非同步的代码。

**总结：**

A. 无论synchronized关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果synchronized作用在一个静态方法或一个类上，则它取得的锁是该类的，该类所有的对象同一把锁。

B. 每个对象只有一个锁（lock）与之相关联，谁拿到这个锁谁就可以运行它所控制的那段代码。

C. 实现同步是要很大的系统开销作为代价的，甚至可能造成死锁，所以尽量避免无谓的同步控制。

**谈谈对重入锁的理解**

可重复获得锁并且不会造成死锁。ReentrantLock和Synchronized都是可重入锁。不同点是前者需要手动释放锁，后者会自动释放。

**volatile的原理**

Volatile的作用是实现可见性（每个线程有自己的内存空间）、防止重排序。

原理

可见性：1，修改了volatile变量时会把修改后的值刷新到内存中。2，修改volatile变量后会导致其他线程中的变量失效，这样就需要再从内存中读取值。

有序性：通过内存栅栏实现顺序的限制。是一种对cpu指令强制要求顺序执行的方法。

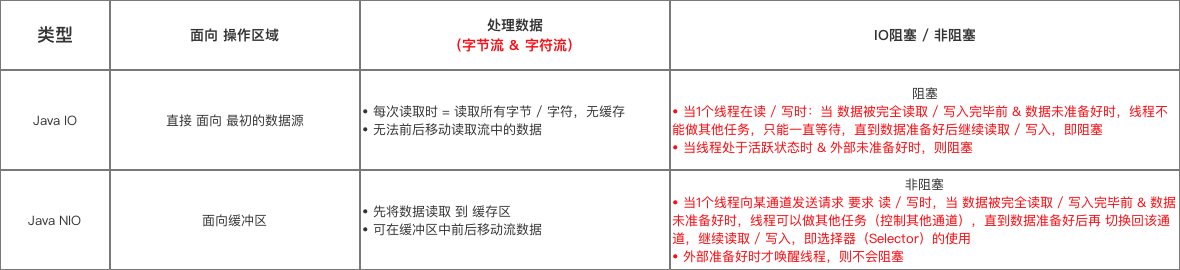
**为什么单例模式下singleton要用volatile修饰？**

实例化对象步骤为1：分配内存空间 2：初始化对象 3：把内存地址赋值给引用，但是由于操作系统可以对指令进行重排序，所以实际步骤不一样的上面的那样，那就可以会导致没初始化好的对象引用暴露出来，所以使用volatile修饰可以防止重排序的发生。

**谈谈volatile关键字的用法**

**谈谈NIO的理解**（netty连接）

基于通道和缓冲区的非阻塞式的IO类（channel、Buffer、Selector），数据通过channel写入缓冲区，从缓冲区通过channel写入，channel就是数据源或者目的。FileChannel fi = fin.getchannel()



**synchronized与ReentrantLock的区别？**

1:前者是jvm层面的，后者是一个java类

2:当线程阻塞时，前者会一直等待；后者可以尝试一定时间后不再等待

3:前者没法判断锁的状态；后者可以

4:前者不可中断，非公平锁；后者可中断，可以是公平锁也可以不是

**ReentrantLock 、synchronized和volatile比较**（把每一种的特点说出来就OK了）

1：synchronized和rentrantlock是锁定变量，只有获得锁的线程可以修改；而volatile可以让多个线程修改变量。

2：volatile只能用来修饰变量

3：volatile不会造成线程阻塞

4：volatile不能保证原子性。

**ReentrantLock的内部实现**（CAS+AQS队列）

先通过CAS尝试获取锁。如果此时已经有线程占据了锁，那就加入AQS队列并且被挂起。当锁被释放之后，排在CLH队列队首的线程会被唤醒，然后CAS再次尝试获取锁。在这个时候，如果：

非公平锁：如果同时还有另一个线程进来尝试获取，那么有可能会让这个线程抢先获取；

公平锁：如果同时还有另一个线程进来尝试获取，当它发现自己不是在队首的话，就会排到队尾，由队首的线程获取到锁。

**死锁的四个必要条件？**（互斥条件、不可剥夺条件、请求与保持条件、循环等待条件）

**怎么避免死锁？**(设置优先级、超时放弃、特定顺序、降低锁的使用粒度，不要几个功能共用一把锁)

**对象锁和类锁是否会互相影响？**（不会）

**什么是线程池，如何使用?**

<https://www.cnblogs.com/superfj/p/7544971.html>

线程池可以将线程进行复用，有工作的时候就从线程池拿来用；没工作的时候就还给线程池。

corePoolSize:线程池核心线程大小。线程池创建的时候线程数为0

maximumPoolSize :线程池允许的最大线程数，他表示最大能创建多少个线程。

keepAliveTime :表示线程没有任务时最多保持多久然后停止。默认情况下，只有线程池中线程数大于corePoolSize 时，keepAliveTime 才会起作用。换句话说，当线程池中的线程数大于corePoolSize，并且一个线程空闲时间达到了keepAliveTime，那么就是shutdown。

### **线程池的任务处理策略：**

如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行处理；

如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

**线程池的关闭**

ThreadPoolExecutor提供了两个方法，用于线程池的关闭，分别是shutdown()和shutdownNow()，其中：

shutdown()：不会立即终止线程池，而是要等所有任务缓存队列中的任务都执行完后才终止，但再也不会接受新的任务

shutdownNow()：立即终止线程池，并尝试打断正在执行的任务，并且清空任务缓存队列，返回尚未执行的任务

**如何选择线程池的大小？**

结合cpu数量考虑，线程池过大过小都不适合。

**如何保证多线程读写文件的安全？**(文件锁)

**多线程断点续传原理**（<https://www.jianshu.com/p/5cf1e6627061>）

RandomAccessFile两个优点：1,既可以读也可以写 2,可以从指定的位置开始读写

**断点续传的实现**

服务端在http请求头中返回一个Content-Range告知待下载的文件的大小，客户端下载了一部分后，网络中断了，待下次网络再次连接的时候，客户端在用RandomAccessFile读取本地已下载的大小，放在请求头中请求续传，服务端后到后就从请求的地方开始续传。续传的状态码是206

**同步工具类**

CountdownLatch() <https://www.cnblogs.com/Lee_xy_z/p/10470181.html>

1：CountDownLatch用来协调多个线程之间的同步，它能够使一个线程等待另外一个或者多个线程完成工作后才继续执行。

原理：使用一个计数器实现的，计数器的初始值和线程数量一样，当一个线程完成自己的任务之后，计数器的值就会减1。当计数器的值变为0的时候，等待的线程就可以恢复执行任务。

比如运动员赛跑，只有等所有的运动员都跑到终点后裁判才开始排成绩。

**CyclicBarrier** <https://www.jianshu.com/p/333fd8faa56e>

CycliBarrier的作用是让所有线程都等待，直到所有线程都完成后再继续下一步执行。

比如去吃饭，有的朋友早到有的朋友晚到，那么就需要等所有人都到齐了大家才开吃。

**Semaphore**

<https://blog.csdn.net/carson0408/article/details/79475723?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.compare&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.compare>

信号量。作用是限制一定数量的线程执行。比如一次只允许10个人进房间，每个人进来前申请一个信号量，当他出去的时候就释放信号量，那么另外的一个人就能进来，所以房间会始终保持有10个人。

#### java中的四种引用的区别以及使用场景

引用强度：强引用 > 软引用（内存不足时） > 弱引用（JVM垃圾回收时） > 虚引用

<https://www.cnblogs.com/pascall/p/10281775.html>

SoftReference<String> softName = new SoftReference<>("张三");内存不足时，软引用会被回收

WeakReference<String> weakName = new WeakReference<String>("hello");无论内存是否足够，都会进行弱引用的回收

虚引用是任何时候都可能被回收，相当于没有

#### 强引用置为null，会不会被回收？

##### （一）Android基础知识点

**四大组件是什么**（Actvitity,Service,BroadcatReceiver,ContentProvider,）

**四大组件的生命周期和简单用法**

**Activity之间的通信方式**

Intent携带bundle数据，通过static变量、通过shareprerence.file或者数据库

**横竖屏切换的时候，Activity 各种情况下的生命周期、**

1：Manifest没有设置configChanges属性

竖屏/横屏启动

onCreate -->onStart-->onResume

切换横屏/竖屏：

onPause-->onSaveInstanceState-->onStop-->onDestroy-->onCreate-->**onStart-->onRestoreInstanceState-->onResume**-->onPause-->onStop-->onDestroy

2：Manifest没有设置configChanges=“orientation”

Android 6.0生命周期与上面的一致

Android 7.0在切换横竖屏的时候比上面的多一个onConfigurationChanged

Android 8.0 在切换横竖屏的时候只走onConfigurationChanged

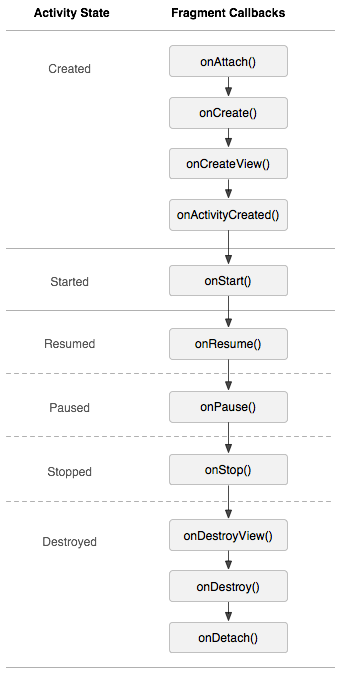
3:AndroidManifest设置了configChanges

android:configChanges="orientation|screenSize"

不管是android 6.0，7.0，8.0都是只走onConfigurations

注：设不设置keyboardHidden对生命周期没有影响

**Activity与Fragment之间生命周期比较**

****

**Activity上有Dialog的时候按Home键时的生命周期**

非activity主题的任何Dialog都不会影响activity的生命周期，所以按下home键的时候跟dialog是没有关系的。onPause()-onStop()

**两个Activity 之间跳转时必然会执行的是哪几个方法？**

注意：如果跳转到一个透明的activity，那么前一个activity只会执行onPause，不会执行onStop方法,所以onStop不一定会执行。

**前台切换到后台，然后再回到前台，Activity生命周期回调方法。弹出Dialog，生命值周期回调方法。**

onPause-onStop-onRestart-onStart-onResume

Dialog非Activity主题的，不影响声明周期

**Activity状态保存与恢复**

在onSaveInstanceSate()中以Bundle数据包的形式保存数据，在onRestoreInstanceSate()或者onCraete()中从Bundle中读取数据进行恢复。

**fragment各种情况下的生命周期**

假设Fragment A要替换Fragment B

1：在activity中用replace替换fragment

1. onPause-onStop-onDestroyView-onDestroy-onDetach
2. onAttach-onCreate-onCreateView-onActivityCreated-onStart-onResume

2:在activity中调用replace和addToBackStack

1. onPause-onStop-onDestroyView
2. (不在backStack中)

onAttach-onCreate-onCreateView-onActivityCreated-onStart-onResume

(B在backStack中)

onCreateView-onActivityCreated-onStart-onResume

**Fragment又该如何保存状态？**

和activity一样，在onSaveInstanceSate()中保存数据，在onRestoreInstanceSate()中恢复

**Fragment状态保存startActivityForResult是哪个类的方法，在什么情况下使用？**

**如何实现Fragment的滑动？（**viewPage）

**fragment之间传递数据的方式？**（好几种方式）

1：先从fragment A用接口传值到Activity中，再从activity中用setArgument传值到fragment B中。

2：在Fragment A中通过fragmentManager的findFragmentById获取到Fragment B的对象，然后用该对象把A的值传过去

3：在Fragment A中通过fragmentManager的findFragmentById获取到Fragment B的对象，然后在A中给B的控件传值。

4：因为所有的fragment都属于一个activity，所以可以通过getActivity获取到activity对象，然后用该对象findViewById找fragment中的控件，给控件赋值。

**为什么不能在Fragment的构造方法中传递参数？**

从源码可以看到fragment是通过反射的方式创建的，那也就意味着它不支持通过构造方法传递参数。fragment是依附于activity上的，当activity为null时，比如横竖屏切换activity重新创建了，那么原来通过构造方法传递到fragment中的值就丢失了。

**Activity 怎么和Service 绑定？**

1：在自定义Service中创建一个内部类继承自Binder

2：在改内部类中创建方法并返回该自定义Service

3：在Activity中通过bindService绑定。在ServiceConnection中获取到Service的实例

**service和activity怎么进行数据交互？**

<https://blog.csdn.net/weixin_41101173/article/details/79718718>

1：如果是通过startService的方式启动的Service,可以直接通过intent.putExtra传递数据给service

2：如果是通过bindService的方式绑定的Service,可以通过自定义binder对象传递参数到service中。或者获取到service的实例，直接调用service中的方法

3：Service可以通过接口回调的方式将数据返回给activity，结合handler来实现前后的数据交互。

4：activity注册广播接收器，service发广播带数据，以此来达到通信的目的。

**Service的开启方式**

startService()，启动完之后该service就在后台运行，其生命周期跟启动它的Context没有任何关系。也不能跟Context通讯。

bindService()启动之后生命周期跟启动它的Context有关，比如Activity、fragment、service等。在Context中解绑之后，如果改Service没有任何绑定后该Service也就结束

**请描述一下Service 的生命周期**

<https://upload-images.jianshu.io/upload_images/944365-34fe9e4c64204f89.png>

**谈谈你对ContentProvider的理解**

<https://www.jianshu.com/p/ea8bc4aaf057>

底层原理是android中Binder机制，用来实现进程间的数据交互和共享

Uri统一资源标识符，外界进程通过URI找到对应的contentProvider和其中的数据，再进行数据的操作。

**说说ContentProvider、ContentResolver、ContentObserver 之间的关系**

外部进程通过ContentResolver统一操作不同的ContentProvider中的数据

当ContentProvider中的数据发生变化的时候，就会触发ContentObserver类

**ContentProvider基本使用步骤**

1：创建数据库类，创建库、表

2：自定义ContentProvider类，重写增删改查的方法并管理数据库类

3：在Manifest中注册ContentProvider，设置权限

（至此，数据提供方的操作就完成了）

4：数据获取方声明权限，访问ContentProvider类操作数据

**请描述一下广播BroadcastReceiver的理解**

用来监听app发出的广播并作出响应，可以用在不同组件间的通信、多线程的通信已经系统发出的广播的监听。采用的是观察者模式，原理如下

(AMS负责四大组件的状态管理)

1：广播接收者通过Binder在AMS注册

2：广播发送者通过Bindler机制向AMS发送广播

3：AMS根据广播要求找到合适的接收者，把广播发送到接受者的消息队列中

4：接收者通过消息循环拿到广播并回调onReceive()方法

**广播的分类**

普通广播、有序广播、本地广播、系统广播

**广播使用的方式和场景**

**在manifest 和代码中如何注册和使用BroadcastReceiver?**

<receiver

//此广播接收者类是mBroadcastReceiver

android:name=".mBroadcastReceiver" >

//用于接收网络状态改变时发出的广播

<intent-filter>

<action android:name="android.net.conn.CONNECTIVITY\_CHANGE" />

</intent-filter></receiver>

**本地广播和全局广播有什么差别？**

**BroadcastReceiver，LocalBroadcastReceiver 区别**

**动态广播最好在onResume中注册，在onPause中注销，因为onPause一定会执行，但是onStop或者onDestroy就不一定**

**静态注册和动态注册的区别**

前者是常驻的，不受任何组件的生命周期的影响，耗电占内存，好处是可以时刻监听广播，在android 8.0后不再允许静态注册广播。

后者跟随组件的生命周期的变化。

**注意**

对于不同注册方式的广播接收器回调OnReceive（Context context，Intent intent）中的context返回值是不一样的：

对于静态注册（全局+应用内广播），回调onReceive(context, intent)中的context返回值是：ReceiverRestrictedContext；

对于全局广播的动态注册，回调onReceive(context, intent)中的context返回值是：**Activity Context**；

对于应用内广播的动态注册（LocalBroadcastManager方式），回调onReceive(context, intent)中的context返回值是：**Application Context**。

对于应用内广播的静态态注册（非LocalBroadcastManager方式），回调onReceive(context, intent)中的context返回值是：**Activity Context**；

**AlertDialog,popupWindow,Activity区别**

AlertDailog是非阻塞式的对话框，弹出时后台还可以做事情。

PopupWindow是阻塞式的对话框，可以展示任意的布局文件，弹出时程序会等待

**Application 和 Activity 的 Context 对象的区别**

(<https://blog.csdn.net/zx_android/article/details/79483748>)

Content的直接子类是ContentIml和ContentWrapper，而ContentWrapper又有三个子类，分别是ContentThemeWrapper、Service和Application。而Actiivity就是继承自ContenThemWrapper。

Application的Content的生命周期是伴随应用的存在而存在的；activity的Content只能存活与当前的activity

应用场景：跟UI相关的上下文都是要使用Activity的Content,其他的都可以。

Content的数量 = Activity的数量+Service的数量+1

**Android属性动画特性**

**LinearLayout、RelativeLayout、FrameLayout的特性及对比，并介绍使用场景。**

(<https://blog.csdn.net/hejjunlin/article/details/51159419>)

从源码中我们似乎能看出，我们先前的测试结果中RelativeLayout不如LinearLayout快的根本原因是RelativeLayout需要对其子View进行两次measure过程。而LinearLayout则只需一次measure过程，所以显然会快于RelativeLayout，但是如果LinearLayout中有weight属性，则也需要进行两次measure，但即便如此，应该仍然会比RelativeLayout的情况好一点。

RelativeLayout的子View如果高度和RelativeLayout不同，则会引发效率问题，当子View很复杂时，这个问题会更加严重。如果可以，尽量使用padding代替margin。

能用两层LinearLayout，尽量用一个RelativeLayout，在时间上此时RelativeLayout耗时更小。另外LinearLayout慎用layout\_weight,也将会增加一倍耗时操作。由于使用LinearLayout的layout\_weight,大多数时间是不一样的，这会降低测量的速度。这只是一个如何合理使用Layout的案例，必要的时候，你要小心考虑是否用layout weight。

**谈谈对接口与回调的理解**

(<https://blog.csdn.net/linshijun33/article/details/48183925?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.nonecase>)

<https://blog.csdn.net/weixin_33972649/article/details/86389129?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.channel_param&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.channel_param>

客户端C调用服务端S中的某个函数A，然后S在合适的时候反过来调用C中的某个函数B，最终相当于C自己调用了自己的方法B，对于C来说，这个B就是回调函数。

但是S并不知道C中的这个B函数叫什么，所以它就定义一个协议，让C提前通过这个协议告诉自己怎么使用B函数，这个协议就是接口。

**写一个回调demo**

**介绍下SurfView**

(<https://www.jianshu.com/p/90a75b9b7115>)

**Android中数据存储方式**

<https://www.jianshu.com/p/967d402d411d>

**GreenDao**

<https://www.jianshu.com/p/9555b8edb878>

只需定义数据模型，GreenDao框架将创建数据对象(实体)和DAO(数据访问对象)，能够节省部分代码。

2.不向性能妥协，使用了GreenDao，大多数实体可以以每秒几千个实体的速率进行插入，更新和加载。

3.GreenDao支持加密数据库来保护敏感数据。

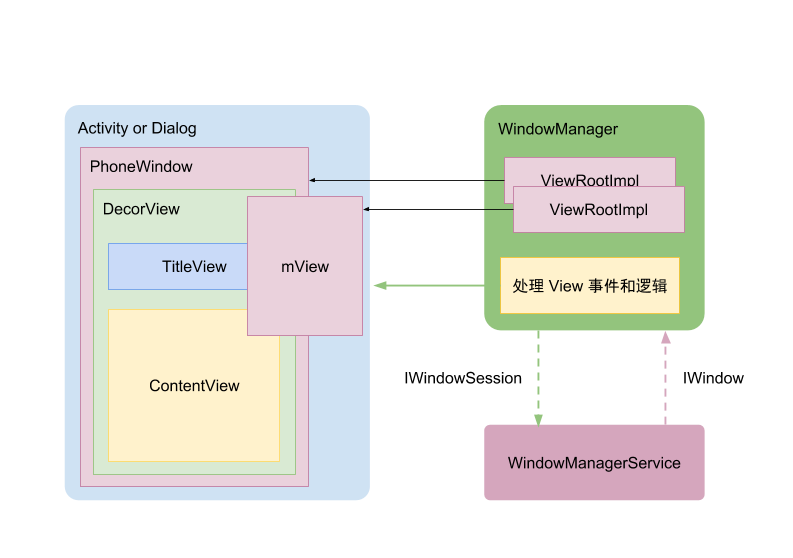
4.微小的依赖库，GreenDao的关键依赖库大小不超过100kb.

#### （二）Android源码相关分析

**Activity-Window-View三者的差别**

<https://blog.csdn.net/freekiteyu/article/details/79408969>

一个 Activity 对应一个 Window 也就是 PhoneWindow，一个 PhoneWindow 持有一个 DecorView 的实例，DecorView 本身是一个 FrameLayout。



**如何优化自定义View**

<https://blog.csdn.net/carson_ho/article/details/56009827>

<https://blog.csdn.net/github_37130188/article/details/89111512>

**低版本SDK如何实现高版本api？**

* <https://blog.csdn.net/ZHENZHEN9310/article/details/90544102?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-3.nonecase>

**HttpUrlConnection 和 okhttp关系**

<https://blog.csdn.net/langtop/article/details/77972855?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-2.nonecase>

**Retrofit的使用步骤**

<https://blog.csdn.net/carson_ho/article/details/73732076>

1：添加retrofit依赖库

2：创建接收服务器返回数据的类

3：创建用于描述网络请求的接口

4：通过Builder创建retrofit的实例（创建实例的配置如下）

5：创建网络请求接口实例并配置请求参数

6：发送网络请求，处理返回的数据(有异步和同步两种请求方式，区别是前者支持线程切换)



**Retrofit 2.0源码解析**

<https://blog.csdn.net/carson_ho/article/details/73732115>

Retrofit是一个restful的Http网络请求框架的封装，网络请求的工作本质上是okhttp完成的。具体过程大概是

1：retrofit将http请求抽象成java接口

2：在接口里面用注解描述配置和网络请求的参数

3：用动态代理的方式将网络请求接口注解解析成http请求

4：最后执行http请求



**Retrofit+Okhttp比较**

Retrofit拥有okhttp优点，性能好，处理快，但是retrofit是高度封装的，解析数据都是用统一的Converter,如果服务器不能给出统一的数据格式,就很难处理。

**图片请求框架Glide**

<https://www.jianshu.com/p/97994c9693f9>

高效的缓存策略，内存开销很小。

**AstncTask+HttpClient 与 AsyncHttpClient有什么区别？**

AsynchttpClient是一个异步网络请求封装库；asynchTask是Android封装好的一个异步任务处理类。

**AsyncTask 如何使用？**

<https://www.jianshu.com/p/ee1342fcf5e7>

1：创建一个AsyncTask的子类并根据需求实现核心方法，onPreExecute等几个方法。

（当这个类被定义为非静态内部类时，可能会引起内存泄漏，最好定义为静态内部类）

2：创建该类的实例对象

3：调用excute方法。

**AsynTask的原理**

Handler+线程池。两个线程池，一个负责任务队列，一个负责执行任务。通过同步锁修饰execute方法，保证任务是串行的。

**LaunchMode应用场景**（就是activity的4种启动模式）

**SpareArray原理**

<https://www.jianshu.com/p/081b78dfe9f6>

使用int和Object类型的数组分别存储key和value，相较于HashMap使用Node，SparseArray在存储单个key-value时更节省内存

SparseArray使用int数组存储int类型的key，避免了int到Integer的自动装箱机制

虽然在存储int到Object映射时的内存使用效率更高，由于使用数组存储数组，在添加或者删除元素时需要进行二分查找，元素较多（超过1000）时效率较低，谷歌给出的建议是数据量不要

**IntentService原理及作用是什么？**

<https://blog.csdn.net/carson_ho/article/details/90437126>

IntentService创建了一个工作线程处理任务。处理完会自动关闭服务。

1：IntentService在onCreate（）中通过HandlerThread开启一个工作线程，并自定义一个handler类，传入子线程的looper,此时handler就绑定到了子线程上。

2：在onStartCommand()调用了onstart()方法，并把intent传入，在onstart方法中将传入的intent通过自定义的handler发送到消息队列中，然后在handlerMessage里面把intent传进onHandleIntent中，随后调用stopSelf()结束服务，这就是为什么IntentService执行完任务会自动停止的原因。

**ApplicationContext和ActivityContext的区别**

**SP是进程同步的吗?有什么方法做到同步？**

**进程和 Application 的生命周期**

<https://blog.csdn.net/github_37130188/article/details/89483117>

**AndroidManifest的作用与理解**

<https://blog.csdn.net/pengliu1998/article/details/77976512>

#### （三）常见的一些原理性问题

**Handler机制和底层实现**

<https://www.jianshu.com/p/b4d745c7ff7a>

<https://www.jianshu.com/p/f0b23ee5a922>

**Handler发送Message的方式：**

1：handler.sendMessage,在使用该方式时，可以自定义内部类继承自Handler，重写handleMessage方法，也可以使用匿名内部类new Handler，本质是相同的，都是创建子类继承handler

2：handler.post(new Runable),无需自定义，直接使用Handler的实例对象调用，并直接在Runable中处理数据。

区别是post方式更新ui的操作可以直接在重写的run方法中定义。不需要创建外部的消息对象，而是根据传图的Runable对象封装消息对象

**Handler的工作原理**

Handler是主线程和子线程之间通信的媒介，负责添加消息到消息队列中并处理分派过来的消息。

Looper是消息队列和handler之间的通信媒介，负责循环从消息队列中取出消息并将消息分发给handler。

工作流程如下：

当主线程创建的时候，天生就带了Looper和MessageQueue,并且Looper会自动进入消息循环，当创建Handler实例的时候，handler就绑定了Looper和MessageQueue。子线程通过handler发送消息到消息队列中，looper循环取出消息队列中的消息并分发给handler，handler处理消息。

**Handler机制如何创建Looper？**

1：Looper.prepareMainLooper()为主线程创建Looper

2：Looper.prepare()为子线程创建Looper,使用Looper.loop()启动消息循环

Looper创建的时候会在构造方法中创建一个对应的MessageQueue

**Handler、Thread和HandlerThread的差别**

<https://www.jianshu.com/p/4a8dc2f50ae6>

**什么是HandlerThread、工作原理是什么？**

是一个封装好的异步类，用于线程间通信的。

原理：继承Thread类，快速创建一个带有Looper的工作线程；通过封装Handle，创建Handler与其他线程的通信。

**HandlerThread使用步骤：**

1：创建handlerThread的实例对象，本质是一个线程

2：启动该线程

3：创建一个Handler实例，并传入handlerThread实例对象获取到的Looper并重写handleMessage方法

4：使用handler实例发送Message

5：使用完毕后，把handlerThrea的实例quit停止掉

**handler发消息给子线程，looper怎么启动？**

**关于Handler，在任何地方new Handler 都是什么线程下?**

New Handler（）传了looper参数的情况下，传入的是哪个线程的looper,handler就在哪个线程下；如果无参构造，哪个线程new Handler（）,就在哪个线程下面。

**请描述一下View事件传递分发机制（和View的绘制机制不一样）**

* <https://www.jianshu.com/p/238d1b753e64>
* <https://www.jianshu.com/p/e99b5e8bd67b>

**Touch事件传递流程**

**事件分发中的onTouch 和onTouchEvent 有什么区别，又该如何使用？**

<https://www.cnblogs.com/aademeng/articles/10922517.html>

onTouch是view的onTouchListener接口中定义的方法，而onTouchEvent是重载的activity方法。当touch事件发生在view上的时候，首先会触发ontouch方法，如果ontouch方法返回true,touch事件将不会传递给activity，也就是onTouchEvent方法将不会被调用。

**View和ViewGroup分别有哪些事件分发相关的回调方法**

**View刷新机制**

**View绘制流程**

* <https://www.jianshu.com/p/c151efe22d0d>

绘制会从ViewRoot开始，从上往下遍视图树，每个view负责自身的绘制，viewGroup还需要负责通知子view进行绘制操作。整个视图绘制操作主要分为mesure、layout和draw,从上往下

**自定义控件原理**

**自定义View如何提供获取View属性的接口？**

**AsyncTask机制**

封装了Handler+Thread，使用线程池管理任务

**AsyncTask原理及不足**

可能会内存泄露、配置改变的时候onPostExecute对新的activity是无效的、后台任务数量有限、只适合执行轻量级的任务，最多几秒钟，不是当成一个异步框架使用

<https://blog.csdn.net/weixin_43507738/article/details/83418989>

**如何取消AsyncTask？**Cancel

**为什么不能在子线程更新UI？**

其实是可以在子线程中更新UI的，而且还不会崩溃，只要没有触发checkThread方法，没判断出来是主线程还是子线程。但是不应该这么写，因为存在崩溃的风险。

<https://www.jianshu.com/p/58c999d3ada7>

**ANR产生的原因是什么？**

**耗时操作或者无响应**

* <https://www.cnblogs.com/dubo-/p/5568730.html>
* ANR定位和修正

**oom是什么？**

Out of memory,主要是对于堆来说的。

**什么情况导致oom？**

<https://www.jianshu.com/p/0616c9134d7a>

**有什么解决方法可以避免OOM？**

**Oom 是否可以try catch？为什么？**

<https://www.jianshu.com/p/f62ab73a7de2>

**内存泄漏是什么？**

**什么情况导致内存泄漏？**

**如何防止线程的内存泄漏？**

**内存泄露场的解决方法**

**内存泄漏和内存溢出区别？**

**非静态内部类可能引起内存泄露实例：**

非静态内部类会隐式的持有外部类的引用

<https://blog.csdn.net/qin19930929/article/details/88909223>

**如何通过广播拦截和abort一条短信？**

只对有序广播有效。

<https://www.jianshu.com/p/f348f6d7fe59>

**广播是否可以请求网络？**

可以，但是要另起线程，而且不能耗时。

**广播引起anr的时间限制是多少？10s**

Activty:5s BroadcastReceiver：前台广播10s,后台广播60s

前台service 20s,后台Service 200s

**计算一个view的嵌套层级**

通过递归调用view.getParents()获取父类布局，并使用一个计数器进行计数。

**Android线程有没有上限？**

有上限，因为系统给每个应用分配的内存是有效的。创建线程也是需要占用内存的。

#### Android为什么引入Parcelable？原理？

Parcel提供了一套机制，可以将序列化后的数据写入到一个共享内存中，其他进程可以通过parcel从这块共享内存中读取字节流，并将它反序列化。

#### 屏幕适配的处理技巧都有哪些?

<https://www.jianshu.com/p/ec5a1a30694b>

布局适配（布局自适应+sw,w,h限定符）、布局组件适配(wrap\_content、match\_parent、weight)、图片资源适配（.9.png+xhdpi）、用户界面适配(不同的界面跳转不同的xml布局)

#### 动态权限适配方案，权限组的概念

* <https://www.jianshu.com/p/07e1a1cff5ad>(动态权限申请框架)
* <https://www.jianshu.com/p/c69ff8a445ed>

#### 下拉状态栏是不是影响activity的生命周期

延伸出Toash,Dialog,自定义Dialog等

<https://www.jianshu.com/p/781bc86f8042>

#### 如果在onStop的时候做了网络请求，onResume的时候怎么恢复？

* **onStop()方法回调之前，系统会自动调用onSaveInstanceState(Bundle outState)**

#### Android中开启摄像头的主要步骤？？？？整理一下

**Socket是什么？**

<https://www.jianshu.com/p/45d27f3e1196>

套接字，是在应用层和tcp/ip协议族中间的一个软件抽象层，是一个封装了TCP/IP协议族的编程接口。

**http协议是什么？**

超文本传输协议，属于应用层；传输效率高，无连接即交换报文前不需要建立http连接、无状态、传输格式简单、可靠性高因为是基于tcp协议建立的；

http协议可以不建立连接，但是需要建立tcp连接

**TCP使用的注意事项,通讯建立的步骤**

<https://www.jianshu.com/p/65605622234b>

步骤：

客户端

1：创建Socket实例，绑定服务端的ip地址和端口号

2：通过socket.getInputStream()获取服务端的消息。

3：通过socket.getOutputStream()发送消息给服务端

4：使用完毕，close资源。

服务端

1：创建ServerSocket实例，绑定端口号

2：每连入一个客户端，通过serversocket.acept()方法创建一个socket,用于与客户端通信。

2：通过socket.getInputStream()获取客户端的消息。

3：通过socket.getOutputStream()发送消息给客户端

4：使用完毕，close资源。

**为什么TCP需要四次握手才能释放？**

Tcp是全双工的，当客户单要求释放连接的时候，service回复ack,仅仅是客户端没有数据要发送了而已，但是服务端可能还有数据要发送到客户端，所以只有服务端也发出连接断开请求的时候才是双边都没有数据要发送了，才能释放。

**为什么TCP是可靠的？**

因为tcp有差错控制机制。

1.出现差错时，让发送方重传差错数据：即 出错重传，采用自动重传机制+超时重传

2.当接收方来不及接收收到的数据时，可通知发送方降低发送数据的效率。拥塞控制

即接收方根据自己缓存的大小，动态调整发送方的发送窗口大小，从未控制发送方的发送速率。

#### Udp使用的注意事项,通讯建立的步骤：

DatagramPackget、DatagramSocket、udp服务器为所有通信使用同一套socket、udp报文所能承载的数据量是65507个字节。

<https://blog.csdn.net/ns_code/article/details/14128987>

<https://www.jianshu.com/p/61de9478c9aa>（udp结合tcp使用）

步骤：

客户端

1：创建DatagramSocket实例，绑定ip地址和端口号，设置阻塞时间等

2：创建DatagramPacket实例，用来接收或者打包数据

3：通过DatagramSocket的实例调用send或者receive方法发送或者接收数据。

服务端：

与客户端基本一致，区别是服务通过send发送数据的时候，需要先从receive的数据里面把客户端的ip地址和端口号读取出来，再给响应的客户端按照这个ip地址和端口发送回去。

#### TCP的3次握手和四次挥手

#### TCP与UDP的区别

1：tcp面向连接，udp无连接。

2：tcp是可靠通讯，保证数据的有序，不丢失，不重复；udp不是可靠通讯

3：tcp传输的是字节流，udp传输的是报文。

4：tcp是一对一通信，udp可以一对多，多对一，一对一

5：tcp占用更多系统资源，udp占用的少

6：tcp的客户端不需要绑定端口号，系统会分配，服务端需要指定端口号。

UDP的客户端可以绑定端口号，也可以不绑定。（注意：tcp和udp可以使用相同的端口号）

#### TCP与UDP的应用

Tcp:适用于准确性要求很高的，比如文件传输、接收邮件等

Udp:适用于效率要求高，但是准确性不需要很高的，比如视频通话、语音通话等等

#### HTTP与HTTPS的区别以及如何实现安全性

https是基于tcp协议之上再加了一层ssl协议的http协议。可以进行加密传输，具有身份认证的网络协议

#### https中哪里用了对称加密，哪里用了非对称加密，对加密算法（如RSA）等是否有了解?

**RSA：**非对称加密算法。乙方生成公钥和私钥，公布公钥，甲方使用公钥加密报文后发送给乙方，乙方使用私钥解密。

**原理：**要把一个很大的整数因数分解成两个有互质关系的因数是极其困难的（两个正整数，除了1以为没有其他的公因子，他们就是互质关系）。

**适用场景：**数据量比较小的，安全性要求极高的。

**DES:**对称加密算法。以秘钥为参数，对明文进行移位、置换压缩等规则生成密文。解密的时候使用秘钥对密文进行逆处理就可得到明文。

**适用场景：**数据量大的

如果想要对一个安全性要求极高但是数据量比较大的数据进行加密，有两种方式，一：将明文分段，逐段使用RSA加密算法加密再传输；二：使用DES整体加密，再用RSA把DES的秘钥加密再传输（https的原理）。

#### 谈谈你对安卓签名的理解。

<https://segmentfault.com/a/1190000012780434>

如果想升级应用，android系统要求新版本和旧版本具有相同的签名与包名，否则拒绝安装。

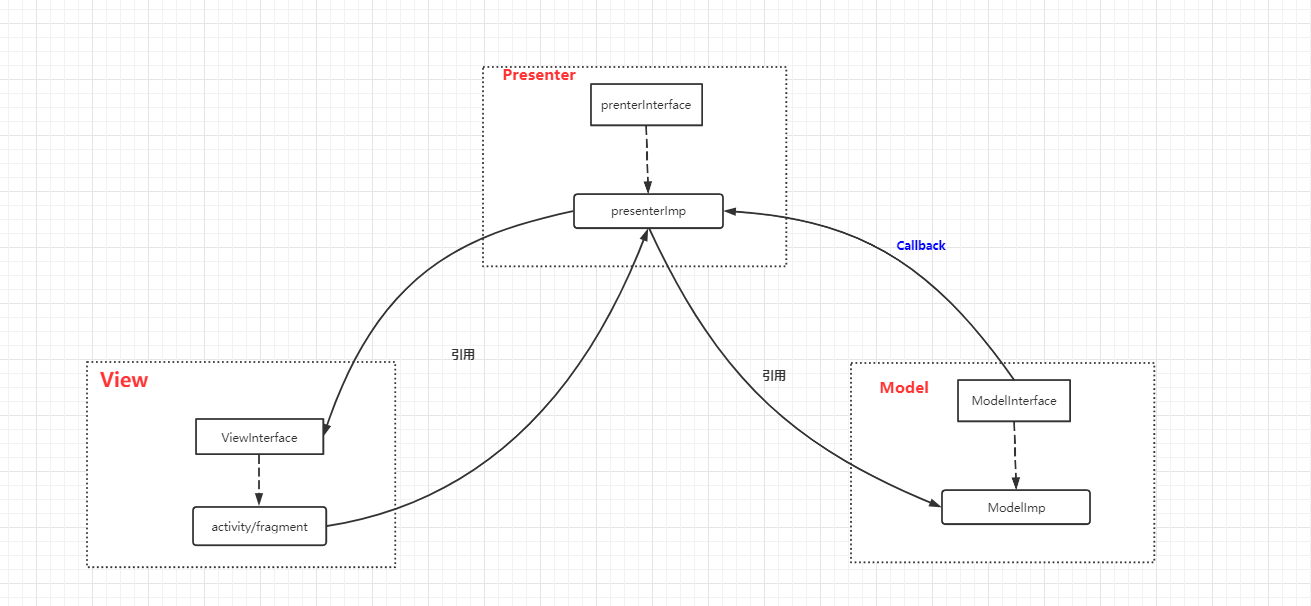
Android系统允许用同一个证书签名的多个应用在一个进程中运行。

可以保证apk的完整性、对身份的鉴定、防止被篡改。

#### （六）架构设计和设计模式

#### 谈谈你对Android设计模式的理解

#### MVC MVP MVVM原理和区别



#### （七）性能优化

#### 如何对Android 应用进行性能分析以及优化?

Cpu profile:查看cpu使用率，内存使用，网络，电池使用，查看内存泄露

使用步骤：

1：运行apk, 打开android profiler，点击 MEMORY

2：点击时间轴，点击下方面板“Arrange by package”

3：查看对象数量，手动点垃圾桶，模拟GC

4：再查看对象的数量，如果数量减少，说明没有内存泄露

5：如果同一对象越来多，则说明内存泄露。

这里要注意单例模式模式引起的内存泄露问题，可以用软引用避免。

#### Java多线程引发的性能问题，怎么解决？

答：线程的西创建和销毁都需要时间，如果大量的线程创建和销毁时，时间消耗就会很明显，导致性能的下降；同时也会消耗cpu和内存，影响系统的吞吐量导致性能下降，甚至会造成OOM;同时还会引起频繁的GC，这样就容易造成内存抖动，直接导致界面卡顿。

#### 启动太慢怎么解决？(Application中不要做复杂和耗时的操作)

#### App启动崩溃异常捕捉

自定义一个异常捕获类实现Thread.UncaughtExceptionHandler

#### 自定义View注意事项

<https://www.jianshu.com/p/146e5cec4863>



关键点：view的结构、view的绘制机制、android的坐标系、view的位置表示、view的角度弧度、颜色的定义、Measure(测量宽高)、layout(计算位置)、draw(绘制视图)的过程等

#### Android资源的访问进制

<https://www.jianshu.com/p/0393e216e428>

<https://blog.csdn.net/buaaroid/article/details/51656373>

#### Android assets和res/raw的区别

两者在打包后都会原封不动的保存在apk中，不会被编译成二进制。

1、 assets目录不会被映射到R中，因此，资源无法通过R.id方式获取，必须要通过AssetManager进行操作与获取；res/raw目录下的资源会被映射到R中，可以通过getResource()方法获取资源。  
2、 多级目录：assets下可以有多级目录，res/raw下不可以有多级目录。

#### 事件分发机制

对于 dispatchTouchEvent，onTouchEvent，return true是终结事件传递。return false 是回溯到父View的onTouchEvent方法。

ViewGroup 想把自己分发给自己的onTouchEvent，需要拦截器onInterceptTouchEvent方法return true 把事件拦截下来。

ViewGroup 的拦截器onInterceptTouchEvent 默认是不拦截的，所以return super.onInterceptTouchEvent()=return false；

View 没有拦截器，为了让View可以把事件分发给自己的onTouchEvent，View的dispatchTouchEvent默认实现（super）就是把事件分发给自己的onTouchEvent。

#### RecyclerView和ListView的性能对比



#### ListView的优化

<https://www.jianshu.com/p/4e8e4fd13cf7>

1:直接在xml中创建ListView组件

2:创建一个activity继承ListActivity

几种适配器的区别：

· ArrayAdapter：简单、易用的Adapter，用于将数组绑定为列表项的数据源，支持泛型

· SimpleAdapter：功能强大的Adapter，用于将XML中控件绑定为列表项的数据源

· SimpleCursorAdapter：与SimpleAdapter类似，用于绑定游标（直接从数据数取出数据）作为列表项的数据源

· BaseAdapter：可自定义ListView，通用用于被扩展。扩展BaseAdapter可以对各个列表项进行最大程度的定制。

BaseAdapter使用步骤

· 定义主xml布局

· 根据需要定义ListView每行所实现的xml布局

· 定义一个Adapter类继承BaseAdapter，重写里面的方法。

· 定义一个HashMap构成的列表，将数据以键值对的方式存放在里面。

· 构造Adapter对象，设置适配器。

· 将LsitView绑定到Adapter上。

#### RecycleView优化

<https://www.jianshu.com/p/c821f9bb370d>

使用步骤：

· 定义主xml布局

· 根据需要定义RecyclerView每行所实现的xml布局

· 定义一个Adapter类继承RecyclerView.Adapter，重写里面的方法。

· 定义一个HashMap构成的列表，将数据以键值对的方式存放在里面。

· 构造Adapter对象，设置适配器。

· 将RecyclerView绑定到Adapter上。

#### ScrollView

<https://www.jianshu.com/p/87a41b8c0dd0>

#### View渲染

#### Bitmap如何处理大图，如一张30M的大图，如何预防OOM

<https://www.cnblogs.com/hzcya1995/p/13309121.html>

Bitmap的用处、存储格式（ARGB）、压缩像素点、Bitmap.Options、Bitmap创建方式、Bitmap回收

避免OOM的方法：

1.我们可以对位图进行压缩，压缩手段有PNG,JPEG,WEBP  
2.对不使用的Bitmap一定要及时回收。  
3.在创建Bitmap时使用try catch步骤OOM异常，使程序更健壮，即使发生了OOM也不会闪退，造成不好的使用体验.

**Android缓存策略**

LruCache、DiskLruCache

<https://www.imooc.com/video/16025>

缓存原理：优先淘汰最近最少使用的缓存对象。使用最少的放在队头，当队列满的时候自动移除队头。

用来缓存图片，以图片的url为key,以图片的bitmap为value进行缓存。

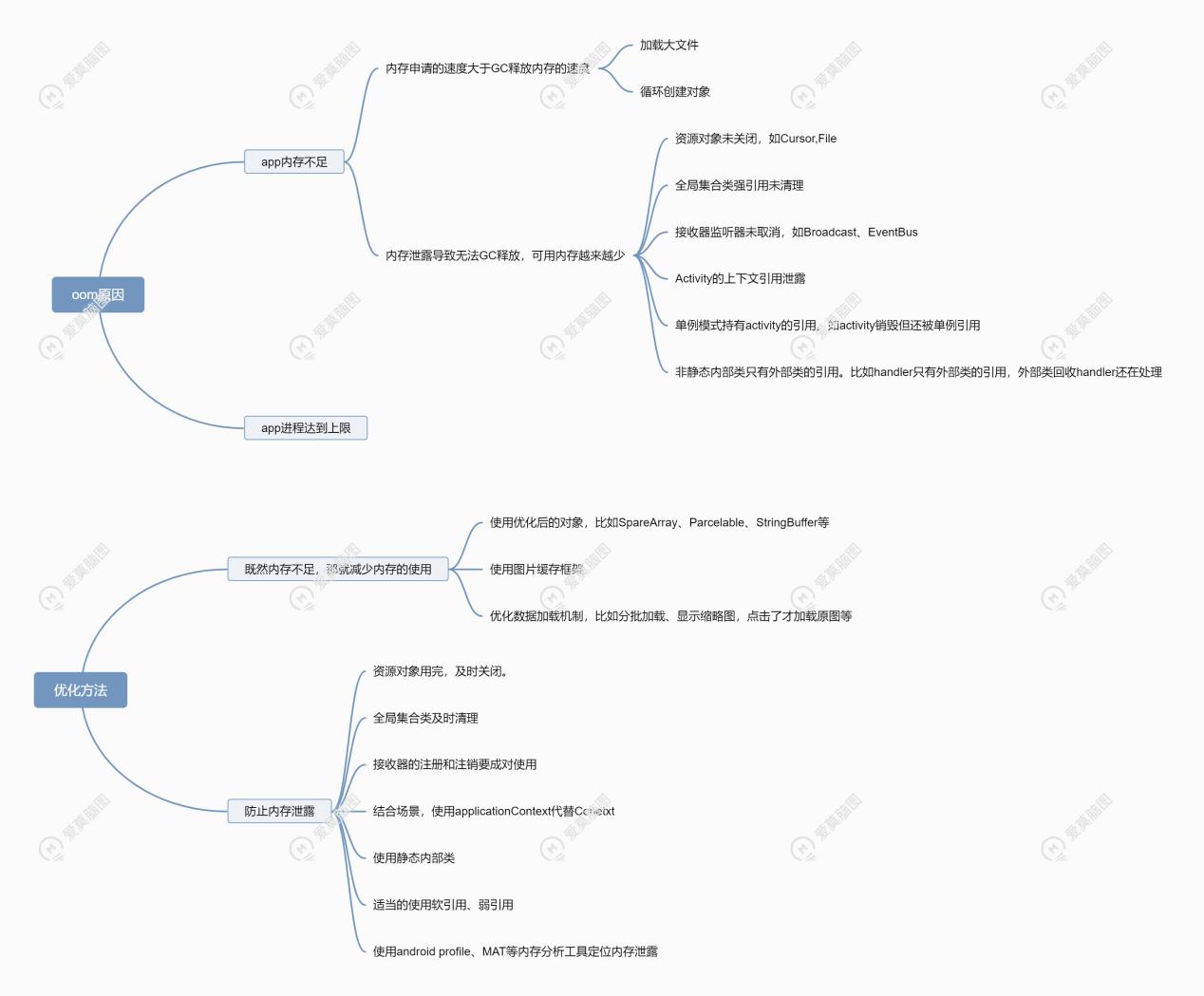
**Android性能优化**

<https://www.jianshu.com/p/516effe7649d>

**OOM的原因以及处理办法**

<https://blog.csdn.net/carson_ho/article/details/79407707>

下面的引起内存泄漏的原因漏了一点：static修饰的成员变量也可能引起内存泄漏

****

#### 

#### （八）NDK、jni、Binder、AIDL、进程通信有关

#### 请介绍一下NDK

#### jni用过吗？

<https://www.jianshu.com/p/16f6a3e3fc45>

#### 如何在jni中注册native函数，有几种注册方式?

静态注册&动态注册

<https://www.jianshu.com/p/1d6ec5068d05>

#### Java如何调用c、c++语言？

#### jni如何调用java层代码？

以上两个看jni api.docx

#### 进程间通信的方式？

<https://www.jianshu.com/p/71dc8fa5210f> Messenger信使，实现跨进程通信

|  |  |  |  |
| --- | --- | --- | --- |
| 方式 | 优点 | 缺点 | 适用场景 |
| Bundle(实现Serialable或者Parcelable) | 简单易用 | 只能传输Bundle支持的数据类型 | 四大组件间的进程通信 |
| 文件共享 | 简单易用 | 不适合高并发，无法即时通讯 | 交换简单的数据，没有即时性要求 |
| AIDL | 强大、可实时通信 | 复杂，序处理线程同步问题 | 复杂的实时性的进程通信 |
| Messenger 信使代码 | 简便，可实时通讯，简版的AIDL | 只支持bundle支持的数据类型 | 低并发，实时性 |
| ContentProvider | 数据访问功能强大，主要提供CRUD操作 | 复杂 | 适合一对多的进程间数据共享 |
| Socket | 功能强大，通过字节流通讯，实时并发 | 繁琐 | 网络数据交换 |

#### Binder机制

<https://blog.csdn.net/carson_ho/article/details/73560642>棒

<https://blog.csdn.net/huachao1001/article/details/51504469?utm_medium=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase&depth_1-utm_source=distribute.pc_relevant_t0.none-task-blog-BlogCommendFromMachineLearnPai2-1.nonecase>

Binder就是一个屏蔽了底层通信细节，让客户端方便调用远程函数的类，使得进程间的通信更加容易，虚拟的设备驱动。

内核空间（共享）+用户空间（私有）

以下是传统跨进程通信的大致过程：

客户端将数据写入到Parcel，调用所持有的Binder引用的函数，该函数将参数、标识符等数据放到客户端的共享内存中，Binder驱动从Client的共享内存中读取数据，根据数据找到对应的远程Service的共享内存，把数据拷贝到Service的共享内存中，并通知远程Service运行某个函数，Service运行完后将结果写到自己的共享内存中，Binder再将结果拷贝到Client的共享内存中，并唤醒Client线程。

Binder的进程间通信过程大致如下：

1：Binder驱动创建一块内核接收缓存区

2：根据Service进程的信息，实现内核缓存区和Service用户空间地址的映射

3：Client发送数据到内核缓存区，由于内核缓存区和Service的用户空间地址存在映射关系，所以也相当于发送数据到了Service的用户空间，从未实现跨进程通信。

#### 什么是AIDL？

Android接口定义语言。

关键点：支持的数据类型、哪些是aidl文件、定向tag、明确导包

#### AIDL如何使用？

<https://github.com/leavesC/IPCSamples/blob/master/note/AndroidIPC%E6%9C%BA%E5%88%B6%EF%BC%883%EF%BC%89-AIDL.md> 代码

Binder池更进一步

**服务端**

1：在服务端定义一个用于在客户端和服务端之间传输的对象，如paramter，并实现parcelable接口用于序列化

2：服务端创建与之对应的aidl文件,parameter.aidl

3：服务端创建AIDL接口,用parameter作为接口的参数，并显式的引入parameter.aidl

4：服务端创建一个Service,在onBind里面实例化AIDL接口，在该接口的方法中即可操作客户端发送过来的对象数据

5：定义并初始化remoteCallbackList,用来管理回调接口的注册

**客户端**

6：把服户端的AIDL文件夹和该序列化对象复制到客户端，并保持路径不变。

7：在客户端通过bindService，根据服务端的包名和Service类的路径绑定服务端的Service，并在客户端ServiceConnect中通过IBinder获取到服务端的AIDL接口对象，并注册其他aidl接口。

8：实例化parameter对象，并给它设置参数

9：通过在第6步中获取到的接口对象，给它设置回调方法，并将parameter传入。

10：在注册的回调函数中获取服务端的返回值。

11：在ondestroy中解绑

**Messenger是什么？使用步骤？**

简版的AIDL,核心也是使用Handler实现的，使用步骤如下：

服务端

1：Service定义一个内部类继承自Handler，并重写handleMessage方法，用来处理客户端发送过来的消息，并可通过调用Message对象的reolyTo方法将数据返回给客户端

2：创建Messeger实例对象，传入步骤一中自定义的handler实例对象

3：在onBind方法中，返回Messeger对象的Binder

客户端

4：通过BindService绑定Service

5：在ServiceConnection中拿到binder对象，通过Messeger实例对象发送消息给Service,并可设置replyTo为客户端的Messeger实例对象，用于消息的回传

6：定义一个内部类继承自Handler重写handleMsg方法用于处理Service发送过来的数据

7：创建步骤5中所说的Messeger对象并传入步骤6中自定义的handler实例对象

总结：Service通过AIDL传一个Messeger对象给客户端，客户端使用该对象发送Message给Serive，Service在自定义的Handler中类中进行消息的处理；反过来，客户端在发送给Service的消息中夹带私货，把自己的Messeger对象也传给Service，Serive载通过该对象发送消息给客户端，客户端同样在自定义的handler类中进行消息的处理

#### 进程和 Application 的生命周期？

onCreate、onTerminate、onLowMemory、onTrimMemory、onConfigurationChanged

#### 谈谈对进程共享和线程安全的认识

关键点：进程内存空间隔离、线程无法同步

#### 谈谈对多进程开发的理解以及多进程应用场景

android:process=”:remote”

<https://www.jianshu.com/p/8ea9738b6df3>

<https://blog.csdn.net/lixpjita39/article/details/77435156>

应用场景：比如要做一个音乐播放器，现在有以下一种方案

1：在activity中播放音乐

2：启动后台service播放音乐

3：启动前台service播放音乐

4：在新进程中启动后台service播放音乐

5：在新进程中启动前台service播放音乐

那么第五种方案是最后的，内存低同时优先级高。多进程的使用场景，比如多模块的应用，当其中的一个进程挂掉后并不影响其他。

#### 开启多进程的方式

1：在Menifest里面给四大组件设置android:process=”xx.xx”属性。

2：通过JNI在native层中fork一个新的进程。

#### 什么是协程？

Kotlin的方法函数，一个线程可以包含多个协程。

<https://www.jianshu.com/p/6dde7f92951e>

#### 内存回收机制、GC回收策略、GC原理时机以及GC对象

关键点:内存模型、回收算法、分代的垃圾回收算法、

本地方法栈

VM栈

方法区

程序计数器

堆

蓝色为线程共享的，橙色为线程私有的。VM栈：存基本数据类型、对象的引用、返回地址等，线程为执行一个方法都会创建一个栈帧，该栈帧会放在VM栈的栈顶，方法执行结束的时候会出栈。本地方法栈就是用来执行本地方法的。程序计数器是用来指示程序运行到哪一行的。堆是线程共享的，存储对象的实例，包括数组，也是GC发生的主要场所。方法区存储虚拟机加载的类信息、常量、静态变量。常量池是方法区的一部分。

回收算法：引用计数法、标记-清除、标记-整理

分代算法：年轻代、年老代，年轻代经过多次Minor GC依然存在的对象将会被放到年老代，年老代满了时候将会发生Full GC

发生Full GC的原因：年老代满了、显示的调用System.gc()

<https://www.cnblogs.com/dz-boss/p/10219503.html>棒