# 个人相关信息

目录

[个人相关信息 1](#_Toc17251)

[简单的介绍下自己 6](#_Toc27208)

[说下你的离职原因 6](#_Toc8669)

[谈谈你对加班的看法 6](#_Toc25727)

[同时来了很多急活 都要你今天完成，怎么办？ 6](#_Toc21997)

[JAVA八股文相关 7](#_Toc13654)

[抽象类和接口的区别 7](#_Toc7407)

[String、StringBuffer、StringBuilder区别 7](#_Toc12755)

[String为什么要设计成不可变的？ 7](#_Toc14473)

[“equals”与“==”、“hashCode”的区别和使用场景？ 8](#_Toc6899)

[深拷贝和浅拷贝的区别 8](#_Toc31749)

[谈谈Error和Exeception的区别 8](#_Toc15371)

[什么是反射机制 8](#_Toc7605)

[static 8](#_Toc28981)

[Final 9](#_Toc13358)

[谈谈你对Java泛型中类型擦除的理解 9](#_Toc32264)

[JAVA集合相关 10](#_Toc18074)

[List Set Map的区别 10](#_Toc7983)

[ArrayList和LinkedList的区别 10](#_Toc20605)

[HashMap与HashTable的区别 10](#_Toc21917)

[谈一谈ArrayList的扩容机制 10](#_Toc23698)

[Arraylist是线程安全的吗 11](#_Toc8296)

[说下HashMap 的实现原理-需要完善 11](#_Toc14258)

[HashMap怎么设置初始值的大小 11](#_Toc17056)

[JAVA-多线程 11](#_Toc1000)

[Java 中使用多线程的方式有哪些 11](#_Toc23693)

[什么是死锁 12](#_Toc29974)

[悲观锁和乐观锁 12](#_Toc7232)

[如何避免死锁 12](#_Toc3039)

[如何实现多线程中的同步 12](#_Toc10082)

[怎么保证线程安全 13](#_Toc2215)

[线程有哪些状态 13](#_Toc12440)

[请谈谈 Thread 中 run() 与 start() 的区别？ 13](#_Toc24565)

[谈一谈线程sleep()和wait()的区别？ 14](#_Toc31998)

[线程池中的七个参数代表什么 14](#_Toc629)

[volatile和synchronized的区别 15](#_Toc645)

[synchronized和Lock的区别 15](#_Toc7784)

[JAVA-JVM 15](#_Toc20024)

[JAVA虚拟机回收相关 15](#_Toc27980)

[垃圾回收的几种算法 16](#_Toc6581)

[Java中有几种引用类型 16](#_Toc30714)

[JVM中类的加载机制与加载过程 17](#_Toc29021)

[类加载机制—双亲委派机制 18](#_Toc19603)

[JAVA-网络 18](#_Toc6401)

[简单介绍一下TCP 18](#_Toc15601)

[览器输入一个地址，按下回车，过程是什么样的 19](#_Toc2852)

[TCP三次握手 19](#_Toc12660)

[为什么要三次握手？ 19](#_Toc3837)

[TCP4次挥手 19](#_Toc15203)

[为什么断开连接需要四次挥手？ 20](#_Toc995)

[Handler 20](#_Toc11070)

[Handler、Looper、MessageQueue、线程的关系 20](#_Toc13517)

[简述Handler的实现原理 20](#_Toc21168)

[Android Handler消息机制？ 21](#_Toc23810)

[一个线程有几个Handler？一个线程有几个Looper？如何保证？ 21](#_Toc5895)

[Handler线程是如何切换的？ 22](#_Toc3898)

[Handler内存泄漏的原因是什么？如何解决？ 22](#_Toc24656)

[主线程为什么不用初始化Looper？ 22](#_Toc31408)

[主线程的Looper死循环为什么不会ANR？ 22](#_Toc4757)

[Handler如何保证MessageQueue并发访问安全？ 23](#_Toc9733)

[Handler的阻塞唤醒机制是怎么回事？ 23](#_Toc27087)

[能不能让一个Message加急被处理？ 23](#_Toc13782)

[什么是Handler同步屏障？ 23](#_Toc5759)

[Activity.runOnUiThread 的理解？ 24](#_Toc17786)

[Handler的post和sendMessage方法的区别和应用场景？ 24](#_Toc10261)

[子线程中创建Handler要调用Looper.prepare() 24](#_Toc19017)

[RecyclerView 24](#_Toc31631)

[RecyclerView和ListView的区别 24](#_Toc32329)

[RecyclerView的四级缓存 25](#_Toc22507)

[滑动10个item，再滑回去，会有几个执行onBindView？ 25](#_Toc30020)

[RecyclerView的复用原理 26](#_Toc28523)

[RecyclerView性能优化。 26](#_Toc22418)

[CachedView和RecycledViewPool的关系 28](#_Toc29627)

[View 28](#_Toc22549)

[自定义View的基本流程 28](#_Toc7284)

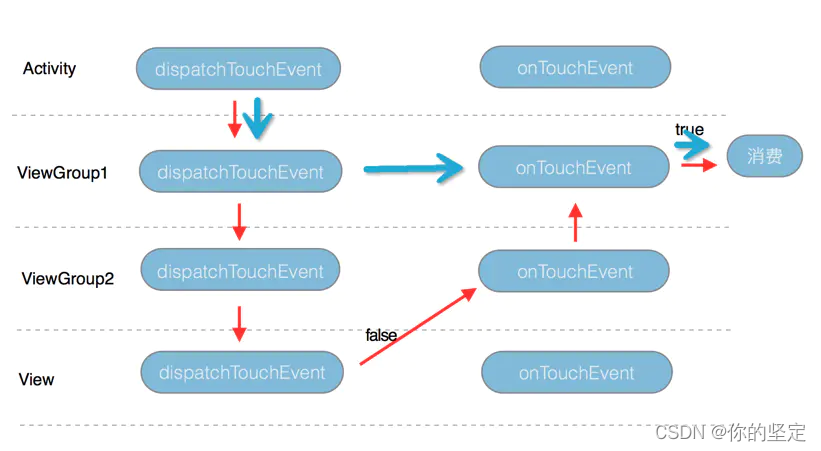
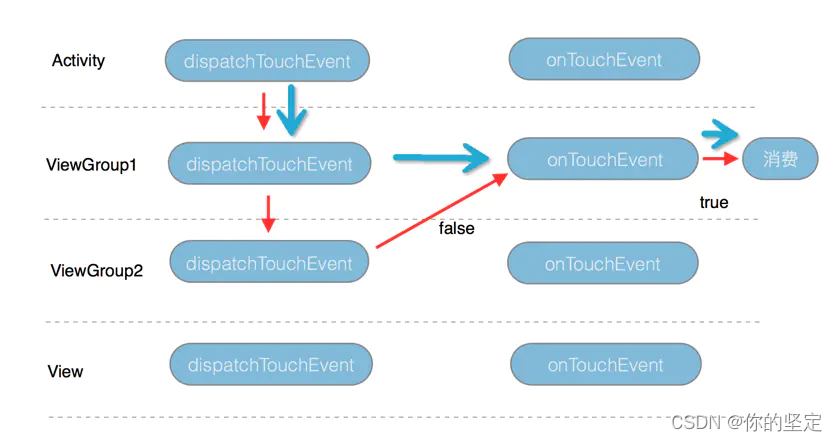
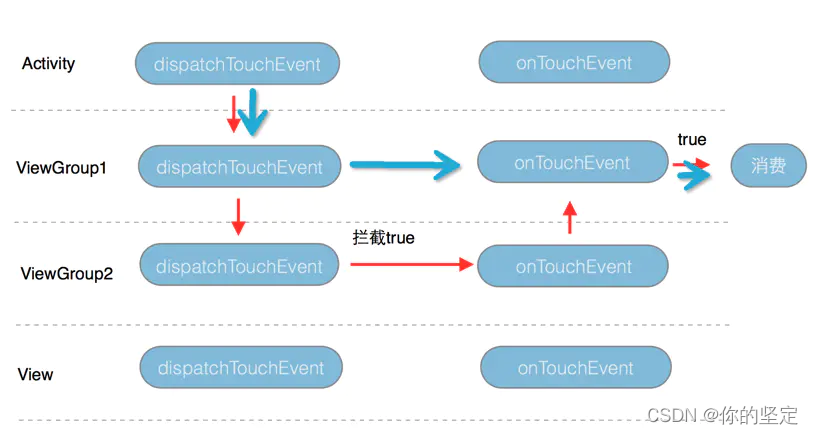
[自定义View的种类 29](#_Toc26431)

[Android Touch事件传递机制 29](#_Toc14999)

[View的事件分发机制 30](#_Toc29462)

[View事件分发伪代码 31](#_Toc2287)

[总结一下同种分发的几条规律 31](#_Toc17558)

[View的绘制流程 32](#_Toc10365)

[实现View滑动有哪些方法 33](#_Toc19679)

[IPC 34](#_Toc29027)

[Android中进程间通信方法 34](#_Toc2318)

[说下Binder基本原理 34](#_Toc17370)

[为什么安卓跨进程通信选择binder机制 34](#_Toc6586)

[Activity 35](#_Toc8769)

[Activity 有几种 launch mode？每一种有什么特点？ 35](#_Toc21346)

[Android 应用冷启动流程 35](#_Toc25676)

[为什么Toast会显示在Activity上面，而不会被Activity覆盖 36](#_Toc30552)

[说下切换横竖屏时Activity的生命周期 36](#_Toc5322)

[Activity和Fragment通信有哪几种方式 37](#_Toc229)

[Fragment向Activity传送数据 37](#_Toc22188)

[Fragment之间互传数据 37](#_Toc4290)

[Service 37](#_Toc21294)

[IntentService与Service的区别 37](#_Toc4373)

[Service的两种启动方式？区别在哪？ 38](#_Toc588)

[Service生命周期 38](#_Toc16017)

[广播 38](#_Toc26609)

[广播有几种注册方式 有什么区别 38](#_Toc26482)

[BroadcastReceiver 与 LocalBroadcastReceiver 有什么区别？ 38](#_Toc29565)

[okhttp 39](#_Toc30562)

[okhttp基本实现原理 39](#_Toc23699)

[okhttp有哪些优点 41](#_Toc24465)

[okhttp框架中对一个主机最大并发数 41](#_Toc28501)

[okhttp框架中有几级请求队列，分别的意义是什么 41](#_Toc9349)

[okhttp中默认有几个拦截器 41](#_Toc2928)

[OkHttp中有哪些设计模式 42](#_Toc26476)

[如果让你设计一个网络请求框架怎么设计 42](#_Toc29871)

[EventBus 42](#_Toc12021)

[Eventbus简介 42](#_Toc18661)

[Eventbus中的设计模式 42](#_Toc5116)

[Eventbus原理 43](#_Toc20082)

[Eventbus粘性事件实现原理 43](#_Toc22669)

[Glide 44](#_Toc10164)

[Gilde:为什么使用Glide? 44](#_Toc25610)

[Glide加载流程 44](#_Toc6935)

[Glide生命周期管理 44](#_Toc13688)

[Glide缓存 45](#_Toc23722)

[Glide对Bitmap是怎么优化的 45](#_Toc9538)

[如果让你设计一个图片请求框架怎么设计 45](#_Toc28005)

[热修复 46](#_Toc23018)

[Sophix：阿里热修复Sophix原理 46](#_Toc1407)

[Tinker：腾讯Tinker热修复原理 46](#_Toc12169)

[Jetpack 47](#_Toc27263)

[Lifecycle是什么，它是如何和Activity或者Fragment结合的 47](#_Toc4109)

[Lifecycle是怎么做到能监听LifecycleOwner（Activity或Fragment）的生命周期的 47](#_Toc22269)

[Lifecycle的实现原理 47](#_Toc27280)

[Lifecycle的状态机说下 48](#_Toc2337)

[什么是LiveData 48](#_Toc10815)

[LiveData的基本实现原理 49](#_Toc30308)

[LiveData递归调用如何容错 50](#_Toc20905)

[LiveData粘性事件和数据倒灌 50](#_Toc22055)

[LiveData粘性事件原理 51](#_Toc526)

[Kotlin 51](#_Toc27098)

[请简述下什么是kotlin？它有什么特性？ 51](#_Toc25167)

[kotlin中with、run、apply、let函数的区别？一般用于什么场景？ 51](#_Toc4115)

[携程 52](#_Toc3407)

[扩展函数原理 53](#_Toc24112)

[MVC MVVM MVP 53](#_Toc15679)

[MVC、MVP和MVVM之间的区别是什么？ 53](#_Toc26792)

[MVC 53](#_Toc5023)

[MVP 54](#_Toc13910)

[MVVM 55](#_Toc8650)

[性能优化 56](#_Toc26964)

[谈谈你对Android性能优化方面的了解？ 56](#_Toc9979)

[Android性能优化主要包括哪些方面？ 56](#_Toc23398)

[内存优化 57](#_Toc14191)

[常见的内存泄漏场景 58](#_Toc13092)

[布局优化？ 58](#_Toc23944)

[网络请求进行优化？ 59](#_Toc24319)

[卡顿优化？ 59](#_Toc3362)

[启动优化？ 60](#_Toc3535)

[绘制优化 60](#_Toc17018)

[WebView进行优化？ 60](#_Toc14488)

[网页调用原生方法大概的原理 61](#_Toc12624)

[说一说Android开发过程中性能优化所用到工具？ 61](#_Toc12193)

[Memory Monitor 61](#_Toc20020)

[LeakCanary 61](#_Toc27924)

[Lint 61](#_Toc2845)

[组件化、模块化、插件化 62](#_Toc16825)

[组件化和模块化的区别 62](#_Toc10672)

[Gradle、多渠道打包 63](#_Toc21654)

[屏幕适配 63](#_Toc22768)

[111 63](#_Toc29812)

[111 63](#_Toc16674)

## 简单的介绍下自己

## 说下你的离职原因

## 谈谈你对加班的看法

首先加班的情况下是分情况的，如果是我自己的本身问题导致的加班，我肯定不希望发生这种事，如果有，我也会利用私下闲余时间去完成任务，但是如果是项目上紧急赶进度需要我配合加班，那我肯定会毫不犹豫配合。

## 同时来了很多急活 都要你今天完成，怎么办？

如果遇到这种问题肯定也不是我一个人决策的，和领导商量，做最紧急的，做不完也会加班赶进度啊

# JAVA八股文相关

## 抽象类和接口的区别

* 抽象类可以提供成员方法的实现细节，接口只能存在public抽象方法
* 抽象类中的成员变量可以是各种不同类型的，接口只能是public static final
* 抽象类可以有构造器、静态代码块和静态方法，接口中则不能
* 一个类只能继承一个抽象类，但是可以实现多个接口
* 抽象类更加侧重于代码的复用，接口更多的是为了约束类的行为

## String、StringBuffer、StringBuilder区别

* String为字符串常量 StringBuffer和StringBuilder为字符串变量，由String创建的字符内容是不可改变的，而由StringBuffer和StringBuidler创建的字符内容是可以改变的。
* StringBuffer是线程安全的，StringBuilder则不是。StringBuffer的所有公开方法都是synchronized修饰的，而StringBuilder并没有
* 效率String<StringBuffer<StringBuilder

String每次连接字符串时都会创建一个新的String对象，所以效率最低

而StringBuffer由于加了synchronized所以次之

## String为什么要设计成不可变的？

* 字符串常量池的需要

当创建一个 String 对象时，如果此字符串已经存在于常量池中，则不会创建一个新的对象，而是引用已经存在的对象

* 允许String对象缓存 HashCode

字符串的不可变性保证了hash码的唯一性，因此可以缓存String的hashCode,这样不用每次去重新计算哈希码。在进行字符串比较时，可以直接比较hashCode,提高了比较性能;

* 安全性

String 被许多的类来当做参数，如 网络url，文件路径 path 等等，如果String 不是固定的，将会引起各种安全隐患

## “equals”与“==”、“hashCode”的区别和使用场景？

* == 对于一个对象来说比较其内存地址是否相同 对于基本数据类型比较值
* equals 一般是字符串的比较其内容是否相同
* hashCode用于辅助比较

## 深拷贝和浅拷贝的区别

这两种拷贝都是针对一个对象来说的

* 浅拷贝

对基本数据类型进行值传递，对引用数据类型进行引用的传递

* 深拷贝

对基本数据类型进行值传递，对于引用数据类型，创建一个新的对象，并且复制其内容

## 谈谈Error和Exeception的区别

* Exception是java程序运行中可预料的异常情况，咱们可以获取到这种异常，并且对这种异常进行业务外的处理。

其中Exeception分为检查性异常和非检查性异常，检查性异常在编写代码的时候必须使用try catch进行捕获，否则编译不通过。非检查性异常则可以通过规范性编写代码进行避免。

* Error是java程序运行中不可预料的异常情况，这种异常发生以后，会直接导致JVM不可处理或者不可恢复的情况。所以这种异常不可能抓取到，比如OutOfMemoryError、NoClassDefFoundError等。

## 什么是反射机制

对于任意一个类，能够知道这个类中的所有属性和方法，对于任意一个对象，能够调用它的任意方法和属性，这种动态获取信息，动态调用对象方法的功能称为JAVA的反射机制。

## static

* static方法
* 静态方法，不依赖任何对象就能访问
* static变量
* 静态变量被所有的对象共享，在内存中只有一个副本，只有类首次加载的时候初始化。
* static代码块
* 可以位于类中的任何地方，类初次加载的时候按照顺序执行。并且只会执行一次

## Final

**final 变量**

final 变量经常和 static 关键字一起使用，作为常量

**Final方法**

代表这个方法不可以被子类的方法重写。

如果你认为一个方法的 功能已经足够完整了，子类中不需要改变的话，你可以声明 此方法为 final。

final 方法比非 final 方法要快，因为在编 译的时候已经静态绑定了，不需要在运行时再动态绑定。

**Final类**

使用 final 来修饰的类叫作 final 类。final 类通常功能是完整的，它们不能被继承。

Java 中有许多类是 final 的，譬如 String，Interger 以及其他包装类。

## 谈谈你对Java泛型中类型擦除的理解

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

# JAVA集合相关

## List Set Map的区别

List存储的数据是有序的，值可以重复

Map存储的数据是无序的 Map的key不能重复 值可以重复

Set 存储的数据是无序的 并且值也不能重复

## ArrayList和LinkedList的区别

ArrayList是基于数组 LinkedList是基于链表

ArrayList适合查询 因为LinkedList要移动指针。

LinkedList适合插入删除 因为ArrayList要移动数据。

## HashMap与HashTable的区别

1 HashMap是继承map接口 HashTable继承自 Dictionary

2 HashMap是非线程安全 HashTable是线程安全的

3 HashMap的key和value都可与允许null；HashTable不允许null

4扩容方式不一样 HashMap默认16 每次扩容\*2，HashTable 初始大小为11 每次扩容2\*n+1;

## 谈一谈ArrayList的扩容机制

ArrayList底层是数组elementData，用于存放插入的数据，

无参构造方法初始化后创建一个空的数组，第一次调用add之后长度扩容为10

有参构造方法则直接创建指定长度的数组。

扩容，当添加元素时，如果元素个数+1> 当前数组长度 【size + 1 > elementData.length】时，进行扩容，扩容后的数组大小是： oldCapacity + (oldCapacity >> 1) 可以理解成1.5倍扩容。

elementData = Arrays.copyOf(elementData, newCapacity);

## Arraylist是线程安全的吗

ArrayList不是线程安全的。

## 说下HashMap 的实现原理-需要完善

HashMap 实际上是一个“链表散列”的数据结构，即数组和链表的结合体。它是基于哈希表的 Map 接口的非同步实现。

数组：存储区间连续，占用内存严重，寻址容易，插入

删除困难；

链表：存储区间离散，占用内存比较宽松，寻址困难， 插

入删除容易；

Hashmap 综合应用了这两种数据结构，实现了寻址容

易，插入删除也容易。

## HashMap怎么设置初始值的大小

如果你在创建的时候没有设置初始值大小，那么它的默认容量是16。

如果你设置了一个初始容量，它会先进行一个判断，判断这个值是不是2的次幂，如果不是将会把容量转化成为2的次幂大小。比如说你设置的容量是27，那么创建的HashMap实际容量是32。

# JAVA-多线程

## Java 中使用多线程的方式有哪些

继承Thread类实现多线程

实现Runnable接口实现多线程

使用ExecutorService、Callable、Future实现有返回结果的多线程

## 什么是死锁

多个线程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进

## 悲观锁和乐观锁

顾名思义，就是比较悲观的锁，总是假设最坏的情况，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会阻塞直到它拿到锁（共享资源每次只给一个线程使用，其它线程阻塞，用完后再把资源转让给其它线程使用）

乐观锁

反之，总是假设最好的情况，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号机制和CAS算法实现。乐观锁适用于多读的应用类型

## 如何避免死锁

**1. 加锁顺序（线程按照一定的顺序加锁）**

确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生。

1. **加锁时限**

线程尝试获取锁的时候加上一定的时限，超过时限则放弃对该锁的请求，并释放自己占有的锁

**3. 死锁检测**

死锁检测是一个更好的死锁预防机制，它主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景。

一个可行的做法是释放所有锁，回退，并且等待一段随机的时间后重试。

一个更好的方案是给这些线程设置优先级，让一个（或几个）线程回退，剩下的线程就像没发生死锁一样继续保持着它们需要的锁

## 如何实现多线程中的同步

Synchronized关键字

Lock锁实现

分布式锁等

## 怎么保证线程安全

* 1，原子性

即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

在Java中，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。

* 2，有序性

即程序执行的顺序按照代码的先后顺序执行。

* 3，可见性

指当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得到修改的值。

当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取共享变量时，它会去内存中读取新值。普通的共享变量不能保证可见性，因为普通共享变量被修改后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。

## 线程有哪些状态

* 第一是创建状态。在创建新的线程对象，没有调用start方法的时候，线程处于创建状态。
* 第二是就绪状态。当调用了线程对象的start方法之后，该线程就进入了就绪状态，此时run方法还没有运行，等待cpu资源
* 第三是运行状态。线程获得cpu资源的支持，此时线程就进入了运行状态，开始运行run函数当中的代码。
* 第四是阻塞状态。由于某种原因导致正在运行的线程让出CPU并暂停自己的执行，即进入堵塞状态。比如调用了wait或者sleep
* 第五是死亡状态。如果一个线程的run方法执行完毕或者调用stop方法后，该线程就会死亡。对于已经死亡的线程，无法再使用start方法令其进入就绪

## 请谈谈 Thread 中 run() 与 start() 的区别？

run() 和普通的成员方法一样，可以被重复调用。但是如果单独调用 run 方法，则不是在子线程中执行。

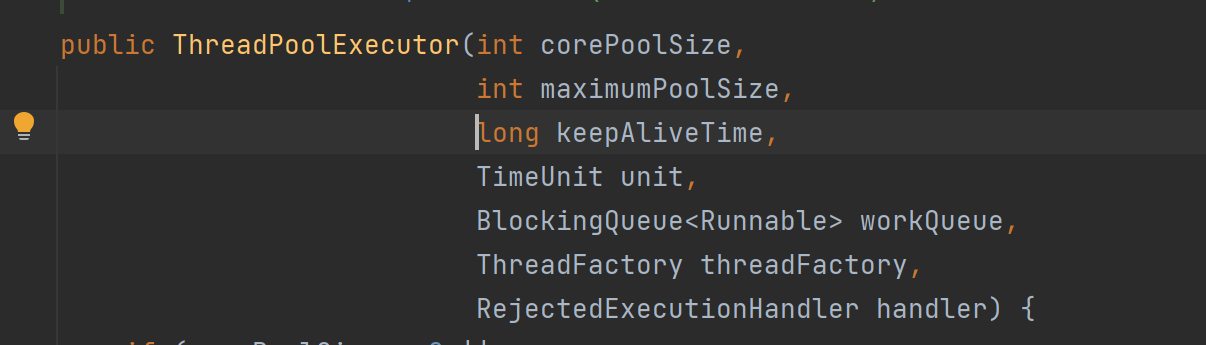
start()这个方法只能被调用一次。调用这个方法后程序会启动一个新的线程来 执行 run 方法。

注意 ：调用start 后， 线程处于可运行状态(并没有运行)，一旦得到 cup 时间片， 就开始执行run 方法，run 方法结束后，线程则立即终止。

## 谈一谈线程sleep()和wait()的区别？

* sleep是Thread类的方法，wait是Object类的方法
* sleep不释放锁，wait释放锁
* sleep不需要Synchronized ，wait需要Synchronized
* sleep不需要唤醒，wait需要唤醒（除wait(int time)）

## 线程池中的七个参数代表什么



* **corePoolSize（核心线程数）**：线程池中保持活动状态的线程数量。当线程池中的线程数少于corePoolSize时，即使有空闲的线程，新任务也会创建新线程来处理。corePoolSize默认情况下会一直保持活动，除非设置了allowCoreThreadTimeOut(true)。
* **maximumPoolSize（最大线程数）：**线程池允许创建的最大线程数。当任务队列已满且正在运行的线程数达到corePoolSize时，新任务会创建新线程，直到达到maximumPoolSize。如果maximumPoolSize为正数且corePoolSize为0，则线程池中的线程数量可以无限制地增长。
* **keepAliveTime（线程空闲时间）：**在线程池中，如果线程数大于corePoolSize，且某个线程空闲时间超过keepAliveTime，那么该线程会被终止，直到线程池中的线程数等于corePoolSize。
* unit（时间单位）：用于指定keepAliveTime的时间单位，可以是秒、毫秒、微秒等。
* **BlockingQueue（任务队列）：**用于保存等待执行的任务的[阻塞队列](https://so.csdn.net/so/search?q=%E9%98%BB%E5%A1%9E%E9%98%9F%E5%88%97&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Matthew93/article/details/_blank)。线程池中的线程会从任务队列中获取任务进行处理。常见的任务队列有LinkedBlockingQueue、ArrayBlockingQueue、PriorityBlockingQueue等。
* **ThreadFactory（线程工厂）：**用于创建新线程的工厂。可以自定义线程的名称、优先级等。
* **RejectedExecutionHandler（拒绝策略）：**在线程池的线程数达到maximumPoolSize且任务队列已满时，新任务无法继续提交。此时拒绝策略会决定如何处理这个任务。常见的拒绝策略有AbortPolicy（直接抛出异常）、CallerRunsPolicy（由提交任务的线程来执行任务）、DiscardPolicy（直接丢弃任务）、DiscardOldestPolicy（丢弃任务队列中最旧的任务）等。

## volatile和synchronized的区别

* volatile关键字解决的是变量在多个线程之间的可见性（一个线程修改的状态对另一个线程是可见的。）；而sychronized关键字解决的是多个线程之间访问共享资源的同步性。
* volatile只能用于修饰变量，而synchronized可以修饰变量，方法，以及代码块。
* volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。
* volatile标记的变量不会被编译器优化；synchronized标记的变量可以被编译器优化

## synchronized和Lock的区别

* synchronized不会引起死锁，而Lock发生异常时，如果没有unLock解锁，可能会引起死锁现象。
* Lock是接口，Sychronsized是关键字；
* 当竞争资源激烈时，Lock性能高于Sychronized。
* Lock可以让等待的锁响应中断，但sychronsized不可以。一直等待下去。

# JAVA-JVM

## JAVA虚拟机回收相关

**怎么找到无用的对象？**

有两种策略（算法）：

引用计数法：

在对象中添加一个计数器，每当有一个地方引用它的时候这个计数器加1，如果引用失效，计数器减一，当计数器为0的时候，则证明这个对象就可以被回收了;缺点，循环引用的问题

可达性分析算法

通过GC ROOTS对象作为起始点，遍历所有的引用链，如果某个对象没有GC ROOTS直接或者间接的链接，则这个对象可以被回收。在JAVA虚拟机中就是通过可达性分析法来判定对象是否存活的

## 垃圾回收的几种算法

* 标记清除

先标记出所有死亡的对象，然后将这些死亡的对象进行清除，缺点是容易产生内存碎片

* 标记整理

先标记出所有存活的对象，然后把这些存活的对象整理到一块连续的区域，然后将这片区域外的空间清除一次。这样不会产生内存碎片，但是需要更新对象的引用

* 标记复制

标记出所有存活的对象，然后将这些对象复制到一个新的内存区域中去，将原来老的内存区域进行清空，缺点是需要比原来更大的内存，也需要更新对象引用

* 分代回收算法

堆分为三代，永久代，新生代，老年代

目前虚拟机使用的算法，它解决了标记整理不适用于老年代的问题，将内存分为不同的年代

在不同的年代使用不同的算法，

新生代存活率低，使用标记复制算法，老年代存活率高，但是没有额外的内存空间对他进行复制，使用使用的是标记整理算法

## Java中有几种引用类型

强引用

当我们使用new创建对象时，被创建的对象就是强引用，如Object object = new Object()，其中的object就是一个强引用了。只要强引用存在，垃圾回收器将永远不会回收被引用的对象，哪怕内存不足时，JVM也会直接抛出OutOfMemoryError，不会去回收。如果想中断强引用与对象之间的联系，可以显示的将强引用赋值为null，这样一来，JVM就可以适时的回收对象了

软引用

软引用是用来描述一些非必需但仍有用的对象。在内存足够的时候，软引用对象不会被回收，只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常。这种特性常常被用来实现缓存技术，比如网页缓存，图片缓存等。在 JDK1.2 之后，用java.lang.ref.SoftReference类来表示软引用。

弱引用

弱引用的引用强度比软引用要更弱一些，无论内存是否足够，只要 JVM 开始进行垃圾回收，那些被弱引用关联的对象都会被回收。在 JDK1.2 之后，用 java.lang.ref.WeakReference 来表示弱引用。如果一个对象只具有弱引用，只要JVM的GC线程检测到了，就会立即回收。弱引用的生命周期要比软引用短很多。不过，如果垃圾回收器是一个优先级很低的线程，也不一定会很快就会释放掉软引用的内存。

虚引用

虚引用是最弱的一种引用关系，如果一个对象仅持有虚引用，那么它就和没有任何引用一样，它随时可能会被回收，在 JDK1.2 之后，用 PhantomReference 类来表示，通过查看这个类的源码，发现它只有一个构造函数和一个 get() 方法，而且它的 get() 方法仅仅是返回一个null，也就是说将永远无法通过虚引用来获取对象，虚引用必须要和 ReferenceQueue 引用队列一起使用。

如果一个对象只具有虚引用，那么它就和没有任何引用一样，随时会被JVM当作垃圾进行GC。

## JVM中类的加载机制与加载过程

虚拟机把描述类的数据从Class文件加载到内存，并对数据 进行校验、转换解析和初始化，最终形成可以被虚拟直接使用的java类型，这就是虚拟机的类加载机制（加载-链接-初始化-使用-卸载）

**类加载的过程**

**加 载**

通过类的全限定名（包名 + 类名），获取到该类的.class文件的二进制[字节流](https://so.csdn.net/so/search?q=%E5%AD%97%E8%8A%82%E6%B5%81&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/hequnwang10/article/details/_blank)将二进制字节流所代表的静态存储结构，转化为方法区运行时的数据结构。在内存中生成一个代表该类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。  
加载二进制数据到内存 —> 映射成jvm能识别的结构 —> 在内存中生成class文件

**链接（连接）**

验 证

确保class文件中的字节流包含的信息，符合当前虚拟机的要求，保证这个被加载的class类的正确性，不会危害到虚拟机的安全。

准 备

为类中的静态字段分配内存，并设置默认的初始值，比如int类型初始值是0。被final修饰的static字段不会设置，因为final在编译的时候就分配了

解 析

解析阶段的目的，是将常量池内的符号引用转换为直接引用的过程（将常量池内的符号引用解析成为实际引用）。

**初始化**

初始化就是执行类的构造器方法init()的过程。  
这个方法不需要定义，是javac编译器自动收集类中所有类变量的赋值动作和静态代码块中的语句合并来的。

若该类具有父类，jvm会保证父类的init先执行，然后在执行子类的init。

使 用

卸载

## 类加载机制—双亲委派机制

jvm对class文件采用的是按需加载的方式，当需要使用该类时，jvm才会将它的class文件加载到内存中产生class对象。

在加载类的时候，是采用的双亲委派机制，即把请求交给父类处理的一种任务委派模式。

* 如果一个类加载器接收到了类加载的请求，它自己不会先去加载，会把这个请求委托给父类加载器去执行。
* 如果父类还存在父类加载器，则继续向上委托，一直委托到启动类加载器：Bootstrap ClassLoader。
* 如果父类加载器可以完成加载任务，就返回成功结果，如果父类加载失败，就由子类自己去尝试加载，如果子类加载失败就会抛出ClassNotFoundException异常，这就是双亲委派模式。

# JAVA-网络

## 简单介绍一下TCP

TCP是一种面向连接的、可靠的、基于[字节流](https://so.csdn.net/so/search?q=%E5%AD%97%E8%8A%82%E6%B5%81&spm=1001.2101.3001.7020)的传输层通信协议

面向连接

发送端和接收端必须建立连接,才能传输数据

传输数据安全,传输速度慢

## 览器输入一个地址，按下回车，过程是什么样的

DNS解析，TCP三次握手，请求报文，响应报文，TCP4次挥手

## TCP三次握手

* 第一次握手，客户端向服务器端发出连接请求，等待服务器确认。
* 第二次握手，服务器端向客户端回送一个响应，通知客户端收到了连接请求。
* 第三次握手，客户端再次向服务器端发送确认信息，确认连接

## 为什么要三次握手？

* **全双工通信**

三次握手是确定通信双方通讯线路是全双工的最小次数，全双工通信是指：通信的双方可以同时发送和接收信息 。

正如双方电话通话：A：喂，能听到吗？B：可以。此时如果A没有反馈，B无法确定A是否能够接收数据。

* **保证可靠性**

另外TCP是可靠传输协议，保证通信的可靠性的手段中包含序列号与确认应答机制。

* **避免资源浪费**

若客户端发出去的第一个连接请求由于某些原因在网络节点中滞留了导致延迟，直到客户端放弃连接后的某个时间点才到达服务端，这是一个早已失效的报文，但是此时服务端仍然认为这是客户端的建立连接请求第一次握手，于是服务端第二次握手回应了客户端。如果没有第三次握手，那么到这里，连接就建立了，但是此时客户端并没有任何数据要发送，会让服务端空等，造成资源浪费。

## TCP4次挥手

* 第一次挥手：主动关闭方发送第一个包，其中FIN标志位为1，发送顺序号seq为X。
* 第二次挥手：被动关闭方收到FIN包后发送第二个包，其中发送顺序号seq为Z，接收顺序号ack为X+1。
* 第三次挥手：被动关闭方再发送第三个包，其中FIN标志位为1，发送顺序号seq为Y，接收顺序号ack为X。
* 第四次挥手：主动关闭方发送第四个包，其中发送顺序号为X，接收顺序号为Y。至此，完成四次挥手。

## 为什么断开连接需要四次挥手？

三次握手是因为建立连接时，ACK和SYN可以放在一个报文里来发送。而关闭连接时，被动关闭方可能还需要发送一些数据后，再发送FIN报文表示同意现在可以关闭连接了，所以它这里的ACK报文和FIN报文多数情况下都是分开发送的。因此断开连接需要4次。

# Handler

## Handler、Looper、MessageQueue、线程的关系

* 一个线程只会有一个Looper对象，所以线程和Looper是一一对应的。
* MessageQueue对象是在new Looper的时候创建的，所以Looper和MessageQueue是一一对应的。
* Handler的作用只是将消息加到MessageQueue中，并后续取出消息后，根据消息的target字段分发给当初的那个handler，所以Handler对于Looper是可以多对一的，也就是多个Hanlder对象都可以用同一个线程、同一个Looper、同一个MessageQueue。

## 简述Handler的实现原理

在Handler的构造函数中，需要获取该线程的Looper对象，然后将Handler与Looper相关联，实现Handler与Looper的绑定。而Looper是与线程绑定的

1）Handler.sendMessage()把消息发送到MessageQueue;

2）MessageQueue.enqueueMessage()进行时间优先级排列；

3）Looper.loop轮询MessageQueue,通过queue.next()获得消息。

4）若 queue.next() == null ,会调用nativePollOnce方法进入阻塞状态，只有当 有消息进入 或 quit 后才会唤醒nativeWake;

5）若 queue.next() != null ,queue.dispatchMessage() 根据 message.mTarget 找到对应的Handler;

6）在Handler.handleMessage()方法中处理message；

## Android Handler消息机制？

作用：

跨线程通信。当子线程中进行耗时操作后需要更新UI时，通过Handler将有关UI的操作切换到主线程中执行。

四要素：

* Message（消息）：需要被传递的消息，其中包含了消息ID，消息处理对象以及处理的数据等，由MessageQueue统一列队，最终由Handler处理。
* MessageQueue（消息队列）：用来存放Handler发送过来的消息，内部通过单链表的数据结构来维护消息列表，等待Looper的抽取。
* Handler（处理者）：负责Message的发送及处理。通过 Handler.sendMessage() 向消息池发送各种消息事件；通过 Handler.handleMessage() 处理相应的消息事件。
* Looper（消息泵）：通过Looper.loop()不断地从MessageQueue中抽取Message，按分发机制将消息分发给目标处理者。

流程

Handler.sendMessage()发送消息时，会通过MessageQueue.enqueueMessage()向MessageQueue中添加一条消息； 通过Looper.loop()开启循环后，不断轮询调用MessageQueue.next()； 调用目标Handler.dispatchMessage()去传递消息，目标Handler收到消息后调用Handler.handlerMessage()处理消息。

## 一个线程有几个Handler？一个线程有几个Looper？如何保证？

可以有多个Handler

只能有一个Looper

因为Looper的构造方法申明为private，只能通过Looper.prepare创建Looper。Looper.prepare方法会先通过ThreadLocal.get检查是否已经创建过Looper，如果不为空，表示已经创建过，则会抛出异常，这样保证了Looper的唯一性。

## Handler线程是如何切换的？

Handler切换线程的方法是：在新线程中创建Handler实例，并绑定到新线程的Looper上。当需要在新线程中执行任务时，可以使用Handler的post()或sendMessage()方法将任务提交给新线程的Looper处理。

## Handler内存泄漏的原因是什么？如何解决？

原因：

Handler被定为为非静态内部类，默认持有外部类的引用(Acivity)

Handler 允许我们发送延时消息，如果在延时期间用户关闭了 Activity，那么该 Activity 会泄露。这个泄露是因为 Message 会持有 Handler，而又因为 Java 的特性，非静态内部类会持有外部类，使得 Activity 会被 Handler 持有，这样最终就 导致 Activity 泄露。

ThreadLocal - Looper - MessageQuene - Message - Handler - Activity

Looper类中：

static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();

解决：

将 Handler 定义成静态的内部类，在内部持有 Activity 的弱引用，并在 Acitivity 的 onDestroy()中调用 handler.removeCallbacksAndMessages(null)及时移除所有消息。

## 主线程为什么不用初始化Looper？

因为ActivityThread中的main已经调用了Looper.prepare

## 主线程的Looper死循环为什么不会ANR？

* 因为当Looper处理完所有消息的时候会进入阻塞状态，当有新的Message进来的时候会打破阻塞继续执行。
* anr定义为5秒钟没有响应触摸事件，或者10秒内广播没有响应。
* 当looper拿不到更多消息时，主线程阻塞并进入休眠nativepullonce，释放CPU资源，当来新消息的时候，通过nativeWark进行唤醒。因此不会anr

## Handler如何保证MessageQueue并发访问安全？

****循环加锁，配合阻塞唤醒机制。****

* 我们可以发现MessageQueue其实是“生产者-消费者”模型，Handler不断地放入消息，Looper不断地取出，这就涉及到死锁问题。如果Looper拿到锁，但是队列中没有消息，就会一直等待，而Handler需要把消息放进去，锁却被Looper拿着无法入队，这就造成了死锁。Handler机制的解决方法是循环加锁。在MessageQueue的next方法中
* 我们可以看到他的等待是在锁外的，当队列中没有消息的时候，他会先释放锁，再进行等待，直到被唤醒。这样就不会造成死锁问题了。
* 那在入队的时候会不会因为队列已经满了然后一边在等待消息处理一边拿着锁呢？这一点不同的是MessageQueue的消息没有上限，或者说他的上限就是JVM给程序分配的内存，如果超出内存会抛出异常，但一般情况下是不会的。

## Handler的阻塞唤醒机制是怎么回事？

****Handler的阻塞唤醒机制是基于Linux的阻塞唤醒机制。****

这个机制也是类似于handler机制的模式。在本地创建一个文件描述符，然后需要等待的一方则监听这个文件描述符，唤醒的一方只需要修改这个文件，那么等待的一方就会收到文件从而打破唤醒。

## 能不能让一个Message加急被处理？

## 什么是Handler同步屏障？

****可以 / 一种使得异步消息可以被更快处理的机制****

* 如果向主线程发送了一个UI更新的操作Message，而此时消息队列中的消息非常多，那么这个Message的处理就会变得缓慢，造成界面卡顿。所以通过同步屏障，可以使得UI绘制的Message更快被执行。
* Message分为3种：普通消息（同步消息）、屏障消息（同步屏障）和异步消息。我们通常使用的都是普通消息，而屏障消息就是在消息队列中插入一个屏障，在屏障之后的所有普通消息都会被挡着，不能被处理。不过异步消息却例外，屏障不会挡住异步消息，因此可以这样认为：屏障消息就是为了确保异步消息的优先级，设置了屏障后，只能处理其后的异步消息，同步消息会被挡住，除非撤销屏障。

## Activity.runOnUiThread 的理解？

这个方法其实相当一是对handler进行封装

首先会判断当前线程是否是在主线程 如果是在主线程 则直接调用这个runnable的run方法

如果不是 就调用当前actvitiy的mHandler的post方法 将这个消息发送到主线程也就是UI线程中处理

## Handler的post和sendMessage方法的区别和应用场景？

其实这两个方法最终都是调用了handler的sendMessageAtTime方法

post的时候 会调用 handler中的getPostMessage方法，通过这个方法 将post的那个runnable对象封装成message对象，将Runnable对象赋值给Message对象中的callback字段

dispatchMessage的时候会对message的callback进行判断 如果不为空 则执handleCallback最终执行行它的run方法

所以区别就是调用post方法的消息是在post传递的Runnabl对象的run方法中处理，而调用sendMessage方法需要重写handleMessage方法或者给handler设置callback，在callback的handleMessage中处理并返回true。

## 子线程中创建Handler要调用Looper.prepare()

因为Handler在初始化的时候会在ThreadLocal中去get一个Looper，在主线程app启动的时候ActivityThread的main方法中已经初始化了一个Looper对象，因此我们不需要自己主动调用Looper.prepare()，在子线程中，没有创建，所以我们需要主动去调用

# RecyclerView

## RecyclerView和ListView的区别

* RecyclerView可以完成ListView,GridView的效果，还可以完成瀑布流的效果。同时还可以设置列表的滚动方向（垂直或者水平）； RecyclerView中view的复用不需要开发者自己写代码，系统已经帮封装完成了。 RecyclerView可以进行局部刷新。 RecyclerView提供了API来实现item的动画效果。
* 在性能上： 如果需要频繁的刷新数据，需要添加动画，则RecyclerView有较大的优势。 如果只是作为列表展示，则两者区别并不是很大。

## RecyclerView的四级缓存

* mAttachedScrap、mChangeScrap(屏幕内)

用于屏幕内itemview快速重用，不需要重新createView和bindView(例如局部刷新)

* mCacheViews(屏幕外)

保存最近移出屏幕的ViewHolder，包含数据和position信息，复用时必须是相同位置的ViewHolder才能复用，应用场景当列表来回滑动时，能直接复用ViewHolder数据，不需要重新bindView。（数据满时候按照先进先出原则）

* mViewCacheExtension(自定义缓存)

不直接使用，需要用户自定义实现，默认不实现。

* mRecyclerPool(缓存池)

当cacheView满了后或者adapter被更换，将cacheView中移出的ViewHolder放到Pool中，放之前会把ViewHolder数据清除掉，所以复用时需要重新bindView。

## 滑动10个item，再滑回去，会有几个执行onBindView？

页面假设可以容纳7条数据

首先，7条数据会依次调用onCreateViewHolder和onBindViewHolder。

往下滑一条（position=7），那么会把position=0的数据放到mCacheViews中。此时mCacheViews缓存区数量为1，mRecyclerPool数量为0。然后新出现的position=7的数据通过postion在mCacheViews中找不到对应的ViewHolder，通过itemtype也在mRecyclerPool中找不到对应的数据，所以会调用onCreateViewHolder和onBindViewHolder方法。

再往下滑一条数据（position=8），如上。

再往下滑一条数据（position=9），position=2的数据会放到mCacheViews中，但是由于mCacheViews缓存区默认容量为2，所以position=0的数据会被清空,然后放到mRecyclerPool缓存池中。而新出现的position=9数据由于在mRecyclerPool中还是找不到相应type的ViewHolder，所以还是会走onCreateViewHolder和onBindViewHolder方法。所以此时mCacheViews缓存区数量为2，mRecyclerPool数量为1。

再往下滑一条数据（position=10），这时候由于可以在mRecyclerPool中找到相同viewtype的ViewHolder了。所以就直接复用了，并调用onBindViewHolder方法绑定数据。

后面依次类推，刚消失的两条数据会被放到mCacheViews中，再出现的时候是不会调用onBindViewHolder方法，而复用的第三条数据是从mRecyclerPool中取得，就会调用onBindViewHolder方法了。

## RecyclerView的复用原理

当RecyclerView要拿一个复用的ViewHolder时

* 如果是预加载，则会先去mChangedScrap中精准查找(分别根据position和id)对应的ViewHolder。
* 如果没有就再去mAttachedScrap和mCachedViews中精确查找(先position后id)是不是原来的ViewHolder。
* 如果还没有，则最终去mRecyclerPool找，如果itemType类型匹配对应的ViewHolder，那么返回实例，让它重新绑定数据。
* 如果mRecyclerPool也没有返回ViewHolder才会调用createViewHolder()重新去创建一个。

## RecyclerView性能优化。

* 提高ViewHolder的复用，减少ViewHolder的创建和数据绑定工作。【最重要】
* 多使用Scrap进行局部更新。使用notifyItemChange、notifyItemInserted、notifyItemMoved和notifyItemRemoved等方法替代notifyDataSetChanged方法
* 使用DiffUtil、SortedList进行局部增量刷新，提高刷新效率。
* 合理设置RecyclerViewPool的大小。如果一屏的item较多，那么RecyclerViewPool的大小就不能再使用默认的5，可适度增大Pool池的大小。如果存在RecyclerView中嵌套RecyclerView的情况，可以考虑复用RecyclerViewPool缓存池，减少开销。
* 为RecyclerView设置setHasStableIds为true，并同时重写RecyclerView.Adapter的getItemId方法来给每个Item一个唯一的ID，提高缓存的复用率。
* 视情况使用setItemViewCacheSize(size)来加大CacheView缓存数目，用空间换取时间提高流畅度。对于可能来回滑动的RecyclerView，把CacheViews的缓存数量设置大一些，可以省去ViewHolder绑定的时间，加快布局显示。
* 当两个数据源大部分相似时，使用swapAdapter代替setAdapter。这是因为setAdapter会直接清空RecyclerView上的所有缓存，但是swapAdapter会将RecyclerView上的ViewHolder保存到pool中，这样当数据源相似时，就可以提高缓存的复用率。
* 优化onBindViewHolder方法，减少ViewHolder绑定的时间。由于ViewHolder可能会进行多次绑定，所以在onBindViewHolder()尽量只做简单的工作。
* 在onBindViewHolder方法中，去除冗余的setOnItemClick等事件。因为直接在onBindViewHolder方法中创建匿名内部类的方式来实现setOnItemClick，会导致在RecyclerView快速滑动时创建很多对象。应当把事件的绑定在ViewHolder创建的时候和对应的rootView进行绑定。
* 数据处理与视图绑定分离，去除onBindViewHolder方法里面的耗时操作，只做纯粹的数据绑定操作。当程序走到onBindViewHolder方法时，数据应当是准备完备的，禁止在onBindViewHolder方法里面进行数据获取的操作。
* 有大量图片时，滚动时停止加载图片，停止后再去加载图片。
* 对于固定尺寸的item，可以使用setHasFixedSize避免requestLayout。
* 优化onCreateViewHolder方法，减少ViewHolder创建的时间。
* 降低item的布局层级，可以减少界面创建的渲染时间。
* Prefetch预取。如果你使用的是嵌套的RecyclerView，或者你自己写LayoutManager，则需要自己实现Prefetch，重写collectAdjacentPrefetchPositions方法。

以上针对RecyclerView的缓存机制展开的优化方案，其实还有几种方案可供参考

* 取消不需要的item动画。具体的做法是：

recyclerView.getItemAnimator()).setSupportsChangeAnimations(false)

* 使用getExtraLayoutSpace为LayoutManager设置更多的预留空间。当RecyclerView的元素比较高，一屏只能显示一个元素的时候，第一次滑动到第二个元素会卡顿，这个时候就需要预留的额外空间，让RecyclerView预加载可重用的缓存。

## CachedView和RecycledViewPool的关系

当一个ItemView被滑动滚出屏幕之后，默认会先被保存在CachedView中。CachedView的默认大小为2，可以通过 setItemViewCacheSize 方法修改它的值。当CachedView已满后，后续有新的ItemView从屏幕内滑出时，会迫使CachedView根据FIFO规则，将之前的缓存的ViewHolder转移到RecycledViewPool中，效果可以参考下图：

# View

## 自定义View的基本流程

* 步骤1：创建自定义View类

首先，需要创建一个自定义View类，该类继承自Android Framework中的现有View类（如View、TextView、ImageView等）。如果需要创建一个全新的View，可以直接继承自View类。

* 步骤2：定义构造函数

自定义View类需要提供三个构造函数，分别接受一个Context对象，一个AttributeSet对象，以及一个由样式属性组成的整数值。这些构造函数在不同的场景下被调用，如在代码中创建View或从XML文件加载View等。

* 步骤3：初始化

在init方法中，可以初始化画笔、颜色、文本等属性。此外，还可以通过AttributeSet对象获取XML布局文件中定义的自定义属性。

* 步骤4：处理自定义属性

如果需要在XML布局文件中使用自定义属性，需要在应用程序的res/values目录下创建一个名为attrs.xml的文件，然后在其中定义属性。

* 步骤5：重写onMeasure方法

为了适应不同的屏幕尺寸和分辨率，需要重写onMeasure方法，以便在需要时测量自定义View的宽度和高度。

int widthMode = MeasureSpec.getMode(widthMeasureSpec);

int widthSize = MeasureSpec.getSize(widthMeasureSpec);

int heightMode = MeasureSpec.getMode(heightMeasureSpec);

int heightSize = MeasureSpec.getSize(heightMeasureSpec);

int measuredWidth = getDefaultSize(getSuggestedMinimumWidth(), widthMeasureSpec);

int measuredHeight = getDefaultSize(getSuggestedMinimumHeight(), heightMeasureSpec);

* 步骤6：重写onDraw方法

最后，需要重写onDraw方法，使用Canvas对象绘制自定义View的内容。在此方法中，可以使用Paint对象设置颜色、样式等属性，并调用Canvas的绘图方法（如drawRect、drawText等）绘制自定义View。

* 步骤7：使用自定义View

完成以上步骤后，即可在布局文件中使用自定义View。只需在XML布局文件中引用自定义View的全类名，并设置相应的属性值。

## 自定义View的种类

对现有View进行扩展 比如直接继承TextView

创建复合控件 比如直接继承ViewGroup

重写View实现全新的控件 继承View

## Android Touch事件传递机制

**在我们点击屏幕时，会有下列事件发生：**

* Activity调用dispathTouchEvent()方法，把事件传递给Window； Window再将事件交给DecorView（DecorView是View的根布局）； DecorView再传递给ViewGroup；
* Activity ——> Window ——> DecorView ——> ViewGroup——> View

**事件分发的主要有三个关键方法**

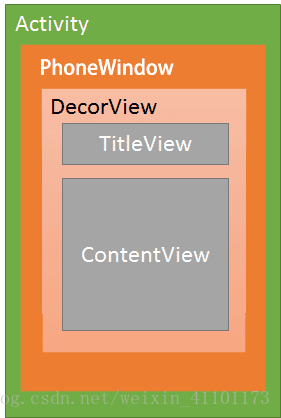
* dispatchTouchEvent() 分发 onInterceptTouchEvent() 拦截 （只有ViewGroup独有此方法）、onTouchEvent() 处理触摸事件
* Activity首先调用dispathTouchEvent()进行分发，接着调用super向下传递
* ViewGroup首先调用dispathTouchEvent()进行分发，接着会调用onInterceptTouchEvent()（拦截事件）。若拦截事件返回为true，表示拦截，事件不会向下层的ViewGroup或者View传递；false，表示不拦截，继续分发事件。默认是false，需要提醒一下，View是没有onInterceptTouchEvent()方法的

**事件在ViewGroup和ViewGroup、ViewGroup和View之间进行传递，最终到达View；**

* View调用dispathTouchEvent()方法，然后在OnTouchEvent()进行处理事件；OnTouchEvent() 返回true，表示消耗此事件，不再向下传递；返回false，表示不消耗事件，交回上层处理。

## View的事件分发机制

Activity -> PhoneWindow ->DecorView(DecorView其实就是一种ViewGroup) ->View



public boolean dispatchTouchEvent（event）

用于进行点击事件的分发

public boolean onInterceptTouchEvent（event）

用于进行点击事件的拦截（VIewGroup）

public boolean onTouchEvent（event）

用于处理点击事件

三个函数的参数均为event，即上面所说的3种类型的输入事件，返回值均为boolean 类型

ViewGroup

dispatchTouchEvent -默认不拦截事件

事件分发的第一个入口

onInterceptTouchEvent

是否拦截事假 true拦截 false不拦截

事件处理 如果这个viewGroup拦截了事件 这个事件就会交给它的onTouchEvent 处理

onTouchEvent

事件处理 true表示到此为止 不向上汇报（L传递）false表示继续向上汇报（U模型）

## View事件分发伪代码





## 总结一下同种分发的几条规律

* 对于activity来说dispatchTouchEvent中无论是返回ture/false都会将事件消费，即不会再往下面传播，只有return super的时候才会传到 ViewGroup()
* 对于dispatchTouchEvent和onTouchEvent来说return false会把该事件交给父容器来处理
* 对于onInterceptTouchEvent来说返回false，顾名思义，该viewGroup将不会拦截该事件，这个事件就可以继续向下传播了，如果onInterceptTouchEvent返回ture就表明该ViewGroup将会拦截该事件，事件将会交给该组件的onTouchEvent来处理，事件也就不会再往下面的组件分发了
* 对于onTouchEvent来说对于传入他的事件，如果返回ture就代表该事件已经被消费，则流程就终止，如果返回false，那么就代表该事件将不会由他处理，将会将给父容器的onTouchEvent来处理。

## View的绘制流程

测量measure，布局layout和绘制draw三个阶段，分别对应了onMeasure,onLayout和onDraw三个方法。

onMeasure

方法会从最外顶层父View依次调用子View的measure方法测量子View大小，最终根据子View大小回调onMeasure测量父View。

onLayout

发放会从最外顶层父View依次调用子View的layout决定位置。

onDraw

方法则是通过canvas对象实现绘制。

## 实现View滑动有哪些方法

* Layout方法（通过计算偏移量，在ACTION\_MOVE中调用layout（+偏移量））
* offSetLeftAndRight和offSetTopAndBottom（加偏移量）
* 动态改变LayoutParams的上下左右margin
* scrollTo和scrollBy方法
* 属性动画

# IPC

## Android中进程间通信方法

使用Bundle

使用[文件共享](https://so.csdn.net/so/search?q=%E6%96%87%E4%BB%B6%E5%85%B1%E4%BA%AB&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/Goals1989/article/details/_blank)

使用Messenger

使用AIDL

使用COntentProvider

使用Socket

## 说下Binder基本原理

通过mmap方式将发送方内核空间与接收方的用户空间进行映射。即用户空间的虚拟内存地址和内核空间的虚拟内存地址指向相同的一块物理地址，这样就只需要将发送方的数据拷贝到内核空间就相当于拷贝到了接收方的用户空间了。

通过binder实现的跨进程通信是c/s模式的，客户端通过远程服务的本地代理向服务端发送请求，服务端处理请求后将结果返回给客户端。

最后说白了aidl的核心就是client调用binder的transact方法，server通过binder的onTransact方法了来执行相应的方法并返回。

## 为什么安卓跨进程通信选择binder机制

* 高性能

从数据拷贝次数来看Binder只需要进行一次内存拷贝，而管道、消息队列、Socket都需要两次，共享内存不需要拷贝，Binder的性能仅次于共享内存。

* 稳定性

Binder基于C/S架构，客户端与服务端彼此独立，稳定性较好。

（上面说到共享内存的性能优于Binder，那为什么不适用共享内存呢，因为共享内存需要处理并发同步问题,容易出现死锁和资源竞争,稳定性较差。）

* 安全性

Android为每个应用分配了UID，作为鉴别进程的重要标志，Android内部依赖这个UID进行权限管理，包括6.0以前的固定权限和6.0以后的动态权限，传统IPC只能由用户在数据包里填入UID/PID，这个标记是在用户空间控制，没有放在内核空间，因此有被恶意篡改的可能，因此Binder的安全性更高



# Activity

## Activity 有几种 launch mode？每一种有什么特点？

* standard模式

activity的默认启动模式，每次会创建一个新的activity实例，依次入栈出栈，满足"先进后出"的原则。

* singleTop模式

 当启动的Activity已经位于栈顶，则直接使用它不创建新的实例。如果没有则创建新的实例。

* singleTask模式

 如果希望Activity在整个应用程序中只存在一个实例，可以使用该模式。每次启动该Activity时，系统首先会检查栈中是否存在该Activity的，如果发现已经存在则直接使用，并将当前Activity之上的所有Activity出栈。

* singleInstance模式

 该模式加载Activity时，无论从哪个任务栈中启动该Activity，只会创建一个Activity实例，并且会使用一个全新的任务栈来装载该Activity实例

## Android 应用冷启动流程

App 启动流程（基于Android8.0）：

* 点击桌面 App图标，Launcher进程采用 BinderIPC（具体为ActivityManager.getService 获取 AMS 实例） 向system\_server的AMS发起 startActivity请求
* system\_server 进程收到请求后，向 Zygote 进程发送创建进程的请求；
* Zygote 进程 fork 出新的子进程，即 App 进程
* App 进程创建即初始化 ActivityThread，然后通过BinderIPC 向 system\_server 进程的 AMS 发起attachApplication 请求
* system\_server 进程的 AMS 在收到 attachApplication 请 求后，做一系列操作后，通知 ApplicationThreadbindApplication，然后发送 H.BIND\_APPLICATION 消息线程收到 H.BIND\_APPLICATION 消息，调用 handleBindApplication 处理后做一系列的初始化操作， 初始化 Application 等
* system\_server 进程的 AMS 在 bindApplication 后，会调用 ActivityStackSupervisor.attachApplicationLocked， 之 后经过一系列操作，在 realStartActivityLocked 方法通过Binder IPC 向 App 进程发送scheduleLaunchActivity 请求；
* App进程的 binder 线程（ApplicationThread）在收到请求后，通过 handler 向主线程发送LAUNCH\_ACTIVITY 消息；
* 主线程收到 message后经过 handleLaunchActivity，performLaunchActivity 方法，然后通过反射机制创建目标Activity；
* 通过 Activityattach方法创建 window并且和 Activity 关 联，然后设置 WindowManager 用来管理 window， 然后通知 Activity 已创建，即调用 onCreate 然后调用 handleResumeActivity，Activity可见

## 为什么Toast会显示在Activity上面，而不会被Activity覆盖

无论toast还是Activity还是dialog，最终都是通过mWindowManager.addView的方式注册到WMS的。

而执行这个方法的时候两个参数，RootView和Params。

WindowManager.LayoutParams中有一个参数为type，这个参数对应的其实就是window显示的优先级，数字越大优先级越高。

## 说下切换横竖屏时Activity的生命周期

**AndroidManifest没有设置configChanges属性竖屏启动：**

竖屏：onCreat->onStart->onResume.

切换横屏时：onPause-> onSaveInstanceState ->onStop->onDestory

onCreat->onStart->onSaveInstanceState->onResume.

但是，我们在如果配置这个属性:android:configChanges="orientation|keyboardHidden|screenSize"  
就不会在调用Activity的生命周期，只会调用onConfigurationChanged方法

## Activity和Fragment通信有哪几种方式

接口方式回调、Bundle通信 、 [eventbus](https://so.csdn.net/so/search?q=eventbus&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/zgy1014/article/details/_blank)、广播、 数据库、文件、SharedPreferences,ContentProvider

## Fragment向Activity传送数据

方法一：使用[java接口](https://so.csdn.net/so/search?q=java%E6%8E%A5%E5%8F%A3&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/m0_53881899/article/details/_blank)（观察者模式）接口回调

方法二：使用第三方组件eventBus、LiveData

## Fragment之间互传数据

Fragment之间无法互传数据，所以需要一个Activity作为中间桥梁辅助两个Fragment之间的数据传输。

# Service

## IntentService与Service的区别

Service->

运行在主线程,用于没有UI工作的任务，由于运行在主线程不能进行耗时操作，需要主动调用stopService()或stopSelf()⽅法关闭自己

IntentService

IntentService继承Service它包含了Service的全部特性，当然也包含service的生命周期，会在执行完所有的请求任务后自行关闭服务，所以我们不必额外调用 stopSelf() 去关闭它，IntentService在执行onCreate操作的时候，内部开了一个工作线程，在工作线程上调用onHandleIntent()，可以执行耗时操作

## Service的两种启动方式？区别在哪？

startService()：开启Service，调用者退出后Service仍然存在。

bindService()：开启Service，调用者退出后Service也随即退出。

## Service生命周期

只是用startService()启动服务：onCreate() -> onStartCommand() -> onDestory

只是用bindService()绑定服务：onCreate() -> onBind() -> onUnBind() -> onDestory

同时使用startService()启动服务与bindService()绑定服务：onCreate() -> onStartCommnad() -> onBind() -> onUnBind() -> onDestory

# 广播

## 广播有几种注册方式 有什么区别

* 静态注册

androidmainfest.xml中注册

常驻型注册，关闭了应用程序依然可以接受到消息

* 动态注册

非常驻型注册，跟随生命周期变化，Activity不可见时取消注册。不过当BroadcastReceiver（receiver）需要更新UI的时候，一般会采用该注册广播的方法

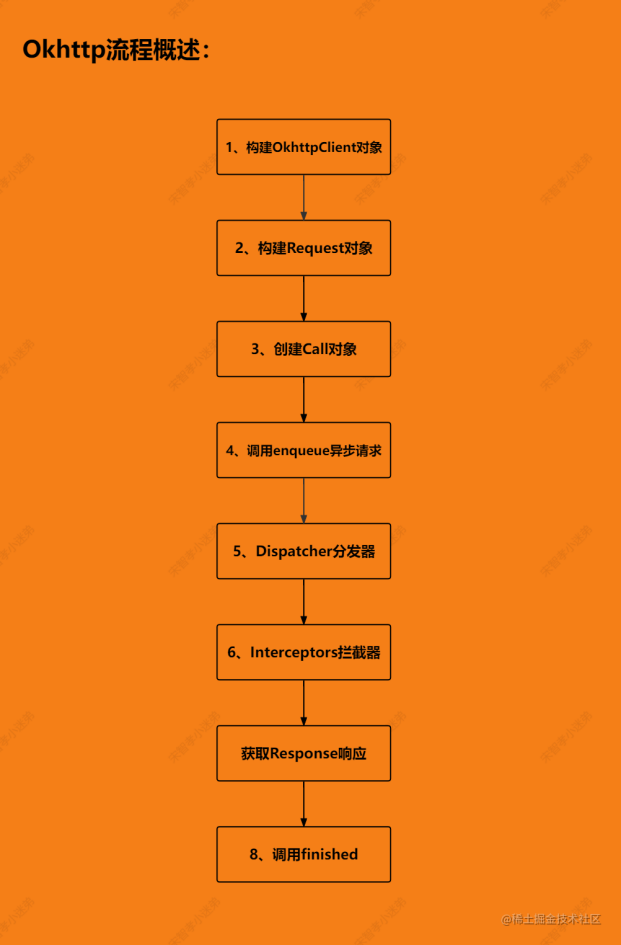
## BroadcastReceiver 与 LocalBroadcastReceiver 有什么区别？

* BroadcastReceiver 是跨应用广播，利用Binder机制实现，支持动态和静态两种方式注册方式。
* LocalBroadcastReceiver 是应用内广播，利用Handler 实现，利用了IntentFilter的match功能，提供消息的发布与 接收功能，实现应用内通信，效率和安全性比较高，仅支持动态注册。

# okhttp

## okhttp基本实现原理

* okhttp主要通过5个拦截器和3个双端队列（2个异步队列，1个同步队列）工作，内部实现通过一个责任链模式完成，将网络请求的各个阶段封装到各个链条中，实现了功能的解耦。
* okhttp底层是通过socket发送http请求与接收响应，但是okhttp实现了连接池的概念，即对于同一个主机的多个请求，可以公用一个Socket连接，而不是每次发送完请求就彻底关闭底层的Socket，这样就实现了链接池的概念，而Okhttp对于socket的读写操作使用okio库进行的封装
* 执行流程
* 通过构建者构建出OkHttpClient对象，在通过newCall方法获得RealCall请求对象
* 通过RealCall发起同步或者异步的请求，决定是同步还是异步是由线程分发器 dispatcher决定的
* 当发起同步请求时会将请求加入到同步队列中依次执行，会阻塞UI线程
* 当发起异步请求的时候会创建一个线程池，并且判断请求队列是否大于最大请求队列64，请求的主机数是否大于5，如果大于就添加到等待队列中，否则就添加到异步执行队列并且执行任务



## okhttp有哪些优点

1，支持http2，对一个域名的所有请求共享同一个Socket

2，内置连接池，支持连接复用，减少延迟

3，支持gzip压缩响应体

4，响应缓存可以完全避免网络重复请求

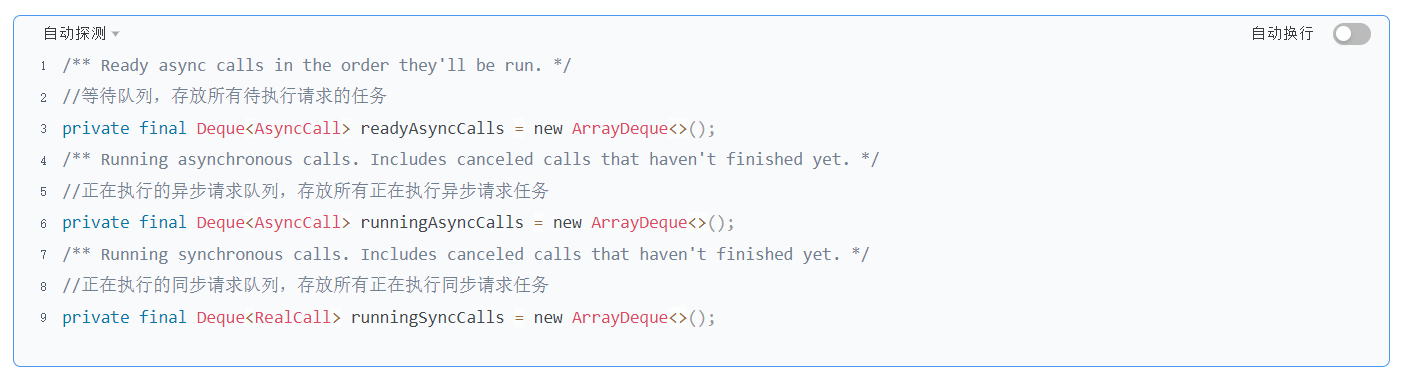
5，请求失败时自动重试主机其他ip，自动重定向

6，丰富的api，可扩展性好

## okhttp框架中对一个主机最大并发数

5个

## okhttp框架中有几级请求队列，分别的意义是什么



## okhttp中默认有几个拦截器

5个

重连和重定向拦截器

请求头拦截器

缓存处理拦截器

连接拦截器

网络请求拦截器

## OkHttp中有哪些设计模式

1.责任链模式，5层拦截器的使用。

2.建造者模式，初始化Client之前配置各种参数。

3.门面模式，OkHttpClient是门面，内部存在大量的接口提供使用。

4.享元模式，OKHttp的链接池复用，避免重复创建。

5.工厂模式，用于生成CacheStrategy对象。

## 如果让你设计一个网络请求框架怎么设计

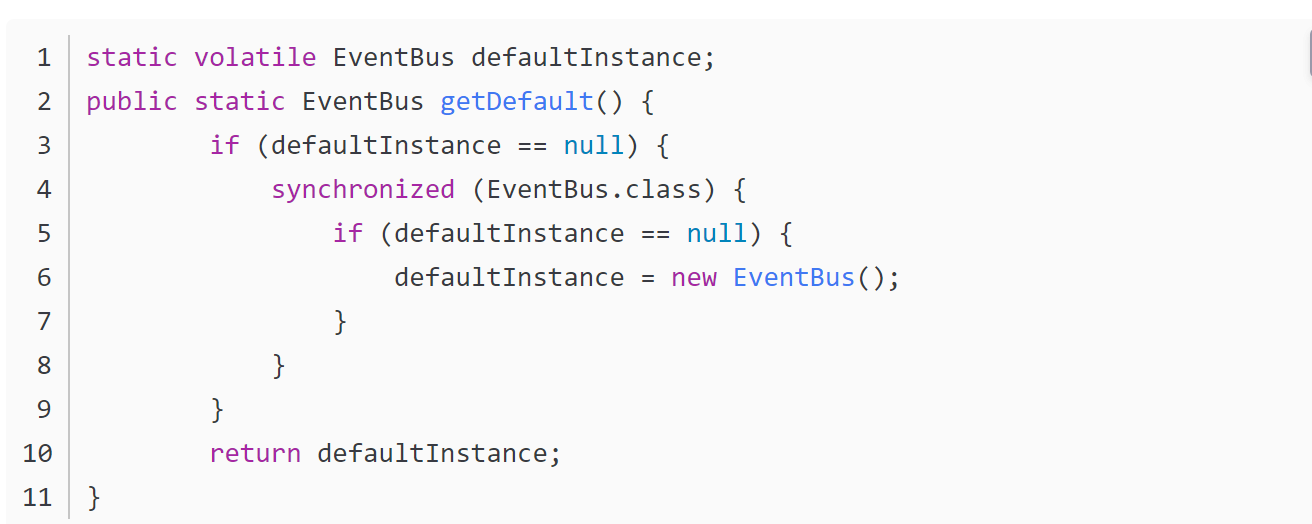
# EventBus

## Eventbus简介

EventBus是一款针对Android优化的发布/订阅（publish/subscribe）事件总线。主要功能是替代Intent,Handler,BroadCast在Fragment，Activity，Service，线程之间传递消息。简化了应用程序内各组件间、组件与后台线程间的通信。优点是开销小，代码更优雅，以及将发送者和接收者解耦。比如请求网络，等网络返回时通过Handler或Broadcast通知UI，两个Fragment之间需要通过Listener通信，这些需求都可以通过EventBus实现。 下面我们就用EventBus来实现以下Fragment之间的数据传递：

## Eventbus中的设计模式

单例模式



观察者模式

当一个MessageEvent对象改变了状态的时候，所有依赖于他的event对象都会收到通知并作出对应的处理。

## Eventbus原理

**注册流程**

首先 eventbus在进行初始化的时候，register方法传入的方法参数为Object，获取该对象的class

然后通过反射的方法得到这个对象和它父类的所有方法

然后将方法的集合进行遍历

判断方法是否是我们加了Subscriber注解的方法 将加了注解的方法存入一个HashMap集合中（会进行方法参数的判断 只要一个参数的方法）

**发送消息流程**

从上序Map中遍历 找到参数一致的方法 通过反射invoke进行方法的调用

这个时候会根据 threadMode进行线程的切换

执行method方法的时候会去获取

注解上标记得线程，然后通过Handler切换到指定线程

## Eventbus粘性事件实现原理

postSticky

EventBus.getDefault().postSticky(Any())

将这个粘性事件添加到了stickyEvents这个Map集合中。之后的逻辑虽多，但和粘性事件没啥关系。

然后在接收的地方注册的时候Eventbus的时候

register

该方法最终会判断当前是否存在注册EventBus前发送的粘性事件，且当前注册类中存在订阅该事件类型的方法，然后立即执行。

# Glide

## Gilde:为什么使用Glide?

**面试官：看你简历上说熟练使用Glide，能够说一说为什么项目上图片加载框架使用的是Glide而不是其它呢？**

* 使用方便,API简洁。with、load、into 三步就可以加载图片
* 生命周期自动绑定，根据绑定的Activity或Fragment生命周期管理图片请求
* 支持多级配置：应用、单独页面（Activity/Fragment）、单个请求进行独立配置。
* 高效缓存策略，两级内存 ，两级文件。
* 支持多种图片格式(Gif、WebP、Video), 扩展灵活

## Glide加载流程

**Glide.with(this).load(url).into(iv)**

Glide的加载过程大致如下，Glide#with获取与生命周期绑定的RequestManager，RequestManager通过load获取对应的RequestBuilder。根据RequestBuilder构建对应的Request,Target 将Request,Target 交给RequestManager进行统一管理。调用RequestManager#track开始进行图片请求。request通过Engine分别尝试从活动缓存、Lru缓存、文件缓存中加载图片，当以上的缓存中都不存在对应的图片后，会从网络中获取。而网络获取大致可以分成，ModelLoader模型匹配，DataFetcher数据获取，然后经历解码、图片变换、转换。如果能够进行缓存原始数据，还会将解码的数据进行编码缓存到文件。

## Glide生命周期管理

通过Glide加载图片会传递一个上下文content 给每个Activity添加一个空白的fragment达到监听Activity生命周期的目的

## Glide缓存

* 活动缓存 ------- 能够看到的图片的缓存
* 内存缓存-------- LRU算法 （Bitmap计数器不为0）
* 磁盘缓存--------Bitmap计数器为0
* 内存缓存
* 活动资源缓存将每一个正在使用的图片加入活动资源缓存，每增加一个使用对象引用计数器加一，否则减一。
* Lru缓存是按照最近最少使用的原则来对图片内存缓存进行维护，当Lru缓存满了的时候，优先移除访问时间最久的那个。
* 活动资源缓存以较小的代价，维护当前在内存中使用的图片资源，减轻Lru缓存的压力，提高缓存效率。
* 文件缓存
* 资源文件缓存是根据当前所需要的资源类型，图片大小等特定信息进行的缓存，当从资源缓存中获取数据的时候，不需要进行解码操作，获取的数据可以直接进行使用。
* 原始数据是根据网络加载的数据，直接进行缓存。使用的时候还需要重新进行解码，转换的流程。

## Glide对Bitmap是怎么优化的

图片的下采样 三级缓存 Bitmap池化重用技术

Glide同样会根据目标控件的尺寸，对图片进行适当的下采样、裁剪和变换，以减少内存占用，并确保加载过程尽快完成。

其下采样的关键实现之一在于DownsampleStrategy类，该类用于指示下采样图像时要使用的算法，从名字上就可以看出，其采用的是23种设计模式中的策略模式，根据不同的缩放模式有不同的策略实现。

## 如果让你设计一个图片请求框架怎么设计

# 热修复

## Sophix：阿里热修复Sophix原理

JAVA方法执行流程

dex文件中体现

调用层调用了一个方法后

执行引擎会通过索引找到方法的ArtMethod

ArtMethod然后找到entry point （方法的首地址指向这块方法的内存）

然后通过这个东西找到了方法的逻辑区域

然后将这个方法压栈到高速缓存区

交给CUP执行

阿里热修复就是通过c代码程序 替换原本有bug的方法的entry point 指针 使其指向修复过的方法的内存首地址

## Tinker：腾讯Tinker热修复原理

安卓中类加载机制

通过PathClassLoader加载dex文件

由于会有多个dex文件 加载完毕会生成一个dexElements数组

程序在实例化一个类的时候，会获取遍历这个数组 去找这个类

dexElements管理了应用所有的dex文件

通过类加载器去加载这个修复包

此时有两个dexElements 当前应用的dexElements 和修复包的dexElements

然后将这两个数组合并

保证修复包的dexElements在当前应用的dexElements前面

合并好之后赋值给当前应用的dexElements

# Jetpack

Lifecycle

## Lifecycle是什么，它是如何和Activity或者Fragment结合的

Lifecycle是一个管理生命周期的工具类，Lifecycle是一个抽象类（实现类LifecycleRegistry），它通过Event枚举类型维护了生命周期分别对应的状态，通过State维护了执行了某一个生命周期函数后和在执行下一个生命周期函数前被观察者（Activity或Fragment）处于什么状态

## Lifecycle是怎么做到能监听LifecycleOwner（Activity或Fragment）的生命周期的

Lifecycle是通过在Activity中绑定了一个空的ReportFragment来实现监听Activity的生命周期的,ReportFragment添加在Activity上之后Activity的生命周期执行的时候会对应的触发ReportFragment的生命周期方法，所以在添加的ReportFragment的生命周期方法中就能监听Activity的生命周期了

## Lifecycle的实现原理

Lifecycle的源码大致分为两部分（LifecycleRegistry主要实现类）：

* addObserver方法为视图控制器注册观察者
* 监听生命周期的变化并通知观察者

**注册流程**

向Activity添加观察者observer，将 state与observer 包装成ObserverWithState类型，state 的初始值为 INITIALIZED ,然后存入集合mObserverMap中，同时通过反射observer类中的带注解的方法放入到androidx.lifecycle.ClassesInfoCache.CallbackInfo mInfo中

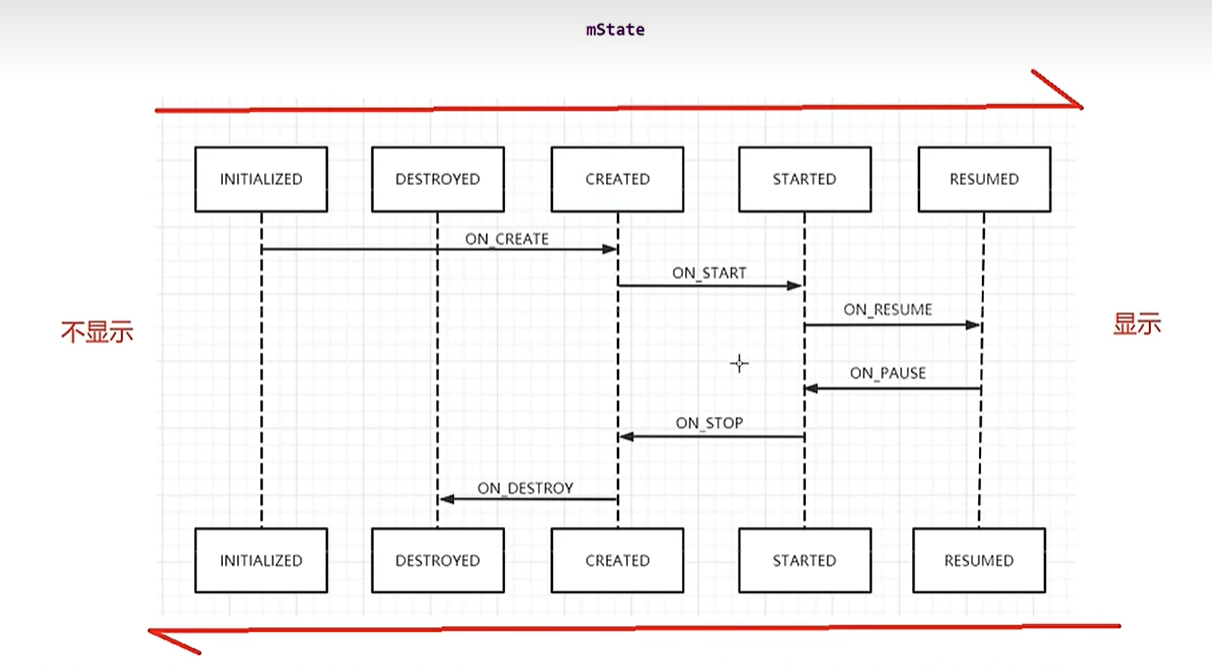
**通知流程**

注册流程走完，Lifecycle 已经持有了所有 observer 的引用(mObserverMap)，Activity的生命周期发生变化，ReportFragment就也会发生变化，在ReportFragment中通过通过调用dispatch(Lifecycle.Event)方法分发的生命周期状态，通过状态机机制，获取了执行了当前生命周期后被观察者处于的newState状态，然后遍历观察者的mObserverMap，然后对比观察者，进行生命周期同步。

observer的 mState 初始状态为INITIALIZED ，然后通过upEvent不断向前分发事件，更新状态，直到 observer 的mState达到当前 LifecycleOwner 的 mState

## Lifecycle的状态机说下

State大小：DESTROYED < INITIALIZED < CREATED < STARTED <  RESUMED



## 什么是LiveData

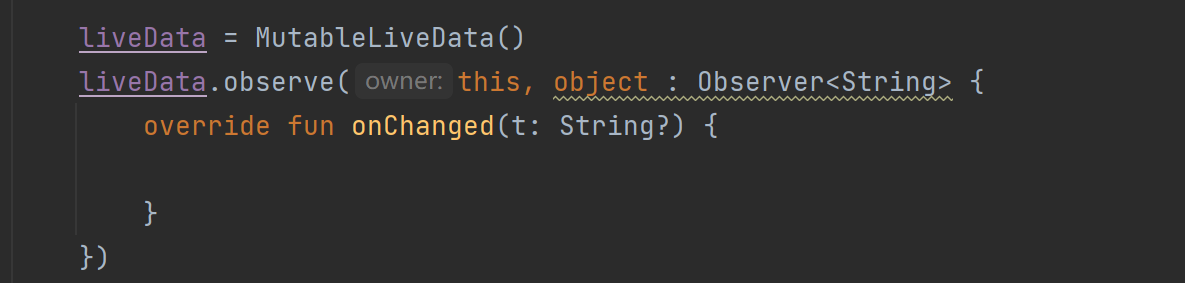
Jetpack LiveData是Android Jetpack中的一个组件，用于帮助开发者实现数据在组件之间的通信，尤其是在视图层和数据层之间。LiveData可以感知组件的生命周期，并且在合适的时机将数据通知到观察者，从而避免了一些潜在的[内存泄漏](https://so.csdn.net/so/search?q=%E5%86%85%E5%AD%98%E6%B3%84%E6%BC%8F&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/wchg21131/article/details/_blank)和空指针异常等问题。

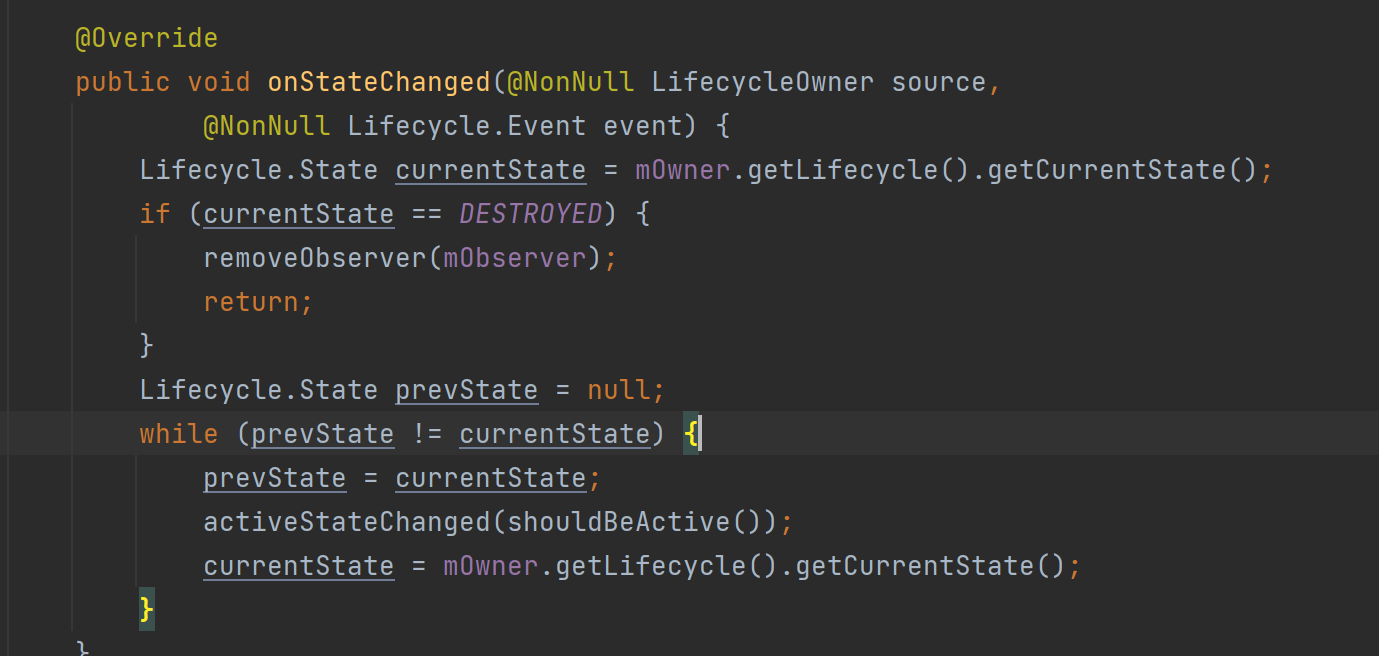
使用LiveData的好处包括：

* 生命周期感知：LiveData可以感知观察者所在的组件的生命周期状态，当对应生命周期为Destroy的时候，会销毁观察者，从而避免潜在的内存泄漏问题。
* 数据通知：LiveData在数据更新时自动通知观察者，从而减少手动更新UI的代码量。
* 数据一致性：LiveData确保观察者只能接收到最新的数据，从而保证数据的一致性。

## LiveData的基本实现原理

**注册流程**





1. 注册的时候主要是通过liveData.observe方法，在这个方法中先判断监听所在宿主生命周期已经结束了，如果结束则不注册
2. 将宿主和监听器进行包装成LifecycleBoundObserver ，这一步是实现反注册、粘性事件、监听器活跃时收到事件的关键（在LifecycleBoundObserver中判断判断当前监听所在宿主的状态是否已经销毁，如果销毁则进行反注册；然后判断Activity的状态是否发生变化（prevState = currentState至少进行一次状态对齐）如果有变化则走事件分发逻辑，这里是粘性事件的关键。

3.将包装好的LifecycleBoundObserver开始监听宿主的生命周期状态。owner.getLifecycle().addObserver(wrapper);

**数据分发流程**

setValue/postValue更新数据

postValue内部通过Handler切换线程，最终也是调用setValue

1.判断当前是否主线程，如果不是则报错(因为多线程会导致数据问题)

2.版本号自增

3.数据源更新

**4.分发事件逻辑**

递归调用容错处理

判断参数中，有没指定Observer，如果有则只通知指定Observer，没有的话则遍历全部Observer通知

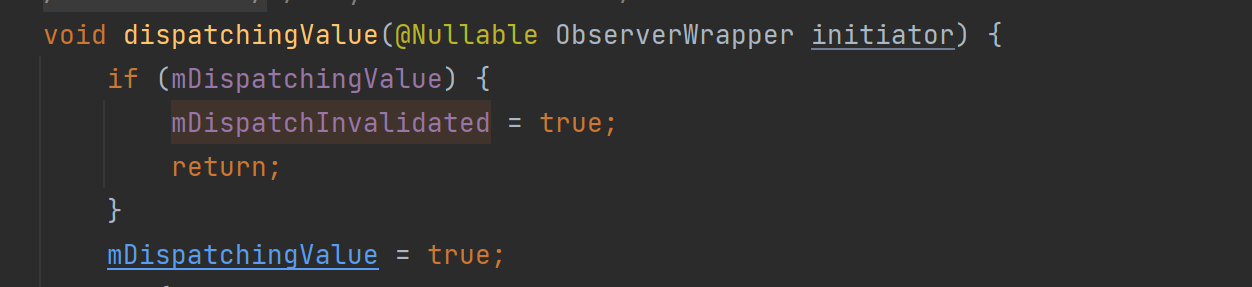
判断是否活跃

判断版本号

对齐版本号

通知监听器

## LiveData递归调用如何容错



通过设置flag标记位 容错，当递归调用的时候，能够让前一次没有跑完的分发事件（循环）进行break掉

## LiveData粘性事件和数据倒灌

粘性事件：先setValue/postValue,后调用observe(),如果成功收到了回调，即为粘性事件。

数据倒灌：先setValue/postValue,后调用observe(new Obs())，至此收到了回调。然后再第二次调用observe(new anotherObs()),如果还能收到第一次的回调，则为“数据倒灌”。

## LiveData粘性事件原理

粘性事件是指先setValue 然后在注册还是能触发接收到值更新的回调

粘性事件主要原来是setValue的时候 会导致liveData的version++

1. 然后到新的activity的时候，注册的时候会判断Activity的状态是否发生变化（prevState = currentState至少进行一次状态对齐）如果有变化则走事件分发逻辑，这里是粘性事件的关键。

# Kotlin

## 请简述下什么是kotlin？它有什么特性？

kotlin是一门编程语言，和java一样都是编译成class文件，然后被虚拟机加载。kotlin是先在android官方优先采用的语言，相比Java，它有以下优势：

* 富有表现力且简洁：您可以使用更少的代码实现更多的功能。表达自己的想法，少编写样板代码。
* 更安全的代码：Kotlin 有许多语言功能，可帮助您避免 null 指针异常等常见编程错误。
* 可互操作：您可以在 Kotlin 代码中调用 Java 代码，或者在 Java 代码中调用 Kotlin 代码。Kotlin 可完全与 Java 编程语言互操作，因此您可以根据需要在项目中添加任意数量的 Kotlin 代码。
* 结构化并发：Kotlin 协程让异步代码像阻塞代码一样易于使用。协程可大幅简化后台任务管理，例如网络调用、本地数据访问等任务的管理。

## kotlin中with、run、apply、let函数的区别？一般用于什么场景？

基本介绍

* with：不是T的扩展函数，需要传入对象进去，不能判空，最后一行是返回值。
* run：是T的扩展函数，内部使用this，最后一行是返回值。
* apply：是T的扩展函数，内部使用this，返回值是调用本身。
* let：是T的扩展函数，内部使用it，当然可以自定义名称(通过修改lambda表达式参数)，最后一行是返回值。
* also：是T的扩展函数，和let一样内部使用it，返回值是调用本身。

使用场景：

用于初始化对象或更改对象属性，可使用apply

如果将数据指派给接收对象的属性之前验证对象，可使用also

如果将对象进行空检查并访问或修改其属性，可使用let

如果想要计算某个值，或者限制多个本地变量的范围，则使用run

## 携程

#### 对于Kotlin中的协程有什么理解？

* 协程视为一种轻量级线程，可用于提高并发代码的性能。
* 它是一种更高效和更简单的方式管理并发的框架，其轻量级线程编写在实际线程框架之上，通过利用函数的协作性质来充分利用它。
* 像是线程：在部分程序执行的过程中，协程的并发执行就是利用的多线程技术（例如：没有进行改版的 Java程序 ）。所以说它像是线程；
* 不是线程：并发任务的调度不是都通过操作系统级别线程切换执行，而是程序本身支持单个线程的多个并发任务。所以也可以说它不是线程，可以叫它们纤程 Fiber ，或者绿色线程 GreenThread 。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。
* 用户态：不是系统级别的线程而且能自主执行异步任务，这种由程序员自己写程序来管理的轻量级线程叫做用户空间线程，具有对内核来说不可见的特性。
* 协作式：要求每个运行中的程序，定位放弃自己的执行权利，让多个任务一起交替执行。它可以执行几行 functionA，然后执行几行 functionB，然后再执行几行 functionA，充分利用线程。其实就是能够动态控制代码在哪个线程里面执行

本质上，协程是轻量级的线程。

但就目前 Kotlin-JVM 而言来说 协程 它就是线程。其本质上还是一套基于原生 Java Thread API 的封装。

* 协程是轻量级的，创建一个线程栈大概需要1M左右，而一个协程栈大概只需要几K或者几十K
* 减少了线程切换的成本，协程可以挂起和恢复，它不会产生额外的开销，由程序自身控制
* 不需要多线程的锁机制：因为只有一个线程，也不存在同时写变量冲突，在协程中控制共享资源不加锁，只需要判断状态就好了，所以执行效率比多线程高很多。

## 扩展函数原理

由反编译的java代码很容易看出，kotlin的拓展函数并没有改变对应类本身的结构，也就是说拓展的类本身并没有真的增加方法。而是增加了了一个方法，将拓展的类对象作为方法的第一个参数传入，然后进行对应的调用。

# MVC MVVM MVP

## MVC、MVP和MVVM之间的区别是什么？

MVC将应用抽象为数据层（Model）、视图层（View）、逻辑层（controller），降低了项目耦合。但MVC并未限制数据流，Model和View之间可以通信。

MVP则限制了Model和View的交互都要通过Presenter，这样对Model和View解耦，提升项目维护性和[模块复用性](https://www.zhihu.com/search?q=%E6%A8%A1%E5%9D%97%E5%A4%8D%E7%94%A8%E6%80%A7&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra={"sourceType":"answer","sourceId":2396493957}" \t "https://www.zhihu.com/question/_blank)。

而MVVM是对MVP的P的改造，用VM替换P，将很多手动的数据=>视图的同步操作自动化，降低了代码复杂度，提升可维护性。

那么什么是MVVM？MVVM是一种软件架构设计模式，它抽离了视图、数据和逻辑，并限定了Model和View只能通过VM进行通信，VM订阅Model并在数据更新时候自动同步到视图。

## MVC

**模型（Model）：**

实体类(数据的获取、存储、数据状态变化)。

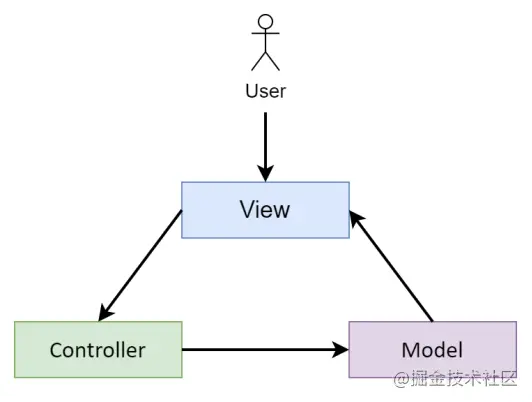
**视图（View）：**

布局文件

**Controller：**

Activity(处理数据、业务和UI)。

**工作原理**



* View接受用户的交互请求。
* View将请求转交给Controller。
* Controller操作Model进行数据更新。
* 数据更新之后，Model通知View数据变化。
* View显示更新之后的数据。

**缺点**

随着界面及其逻辑的复杂度不断提升，Activity类的职责不断增加，以致变得庞大臃肿。

为了解决MVC的缺点，MVP 框架被提出来。

## MVP

MVP（Model-View-Presenter）模式是一种改进的 MVC 模式，它将控制器（Controller）替换为了 Presenter。Presenter 旨在将模型（Model）和视图（View）分离，并将逻辑放在 Presenter 中。在 MVP 中，视图是被动的，它只负责展示数据和接收用户输入，而 Presenter 则负责响应用户输入、更新数据和控制视图状态。

**模型（Model）：**

实体类(数据的获取、存储、数据状态变化)。

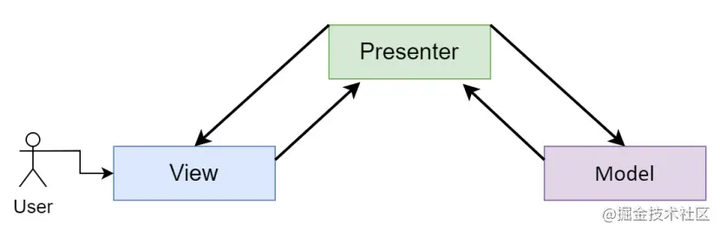
**视图（View）：**

布局文件+Activity。

**Presenter：**

中介，负责完成View与Model间的交互和业务逻辑。

**工作原理**



* View 接收用户交互请求
* View 将请求转交给 Presenter(V调用P接口)
* Presenter 操作Model进行数据更新(P调用M接口)
* Model 通知Presenter数据发生变化(M调用P接口)
* Presenter 更新View数据(P执行接口,V相应回调)

**MVP 模式的优点如下**

* 复杂的逻辑处理放在Presenter进行处理，减少了Activity的臃肿。
* 解耦。Model层与View层完全分离，修改V层不会影响M层，降低了耦合性。
* 可以将一个Presenter用于多个视图，而不需要改变Presenter的逻辑。
* Presenter层与View层的交互是通过接口来进行的，便于单元测试。

**MVP 模式也有一些缺点**

它需要编写更多的代码，因为需要编写 Presenter 类来处理逻辑。

它需要更多的时间和精力来设计和实现，因为需要考虑如何分离视图、模型和 Presenter。

维护困难。Presenter中除了业务逻辑以外，还有大量的View->Model，Model->View的手动同步逻辑，造成Presenter比较笨重，维护起来会比较困难。

## MVVM

**Model：**

实体类(数据的获取、存储、数据状态变化)。

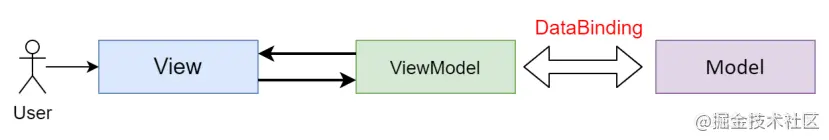
**View：**

布局文件+Activity。

**ViewModel：**

关联层，将Model和View进行绑定，Model或View更改时，实时刷新对方。

**工作原理**



* View 接收用户交互请求
* View 将请求转交给ViewModel
* ViewModel 操作Model数据更新
* Model 更新完数据，通知ViewModel数据发生变化
* ViewModel 更新View数据

**MVVM 模式的优点如下**

它可以将应用程序的不同部分分离开来，使得代码更加模块化和可维护。

它可以提高代码的可测试性，因为它将逻辑分离到 ViewModel 层，使得测试更加容易。

它可以提高开发效率，因为 ViewModel 可以管理多个视图，并且可以处理用户输入和数据更新，减少了视图和模型之间的交互。

**MVVM 模式的缺点如下**

它增加了代码的复杂性，因为需要创建额外的类来处理视图和模型之间的通信。

它需要更多的学习和了解，因为它需要理解数据绑定和 LiveData 等概念

# 性能优化

## 谈谈你对Android性能优化方面的了解？

## Android性能优化主要包括哪些方面？

* 布局优化
* 安装包优化
* 网络优化
* 内存优化
* 卡顿优化
* 启动优化
* 绘制优化

## 内存优化

答：Android中的内存优化总的来说就是开源和节流，开源就是扩大内存，节流就是避免内存泄漏，内存泄漏的本质就是较长生命周期的对象引用了较短生命周期的对象。内存泄漏原因：堆上分配的对象已经不会再使用，但是GC收集器无法对其进行回收，此对象被强应用所引用 。

* 1 静态变量导致的内存泄漏

解决办法：将内部类设为静态内部类或独立出来；使用context.getApplicationContext()。

* 2 单例模式导致的内存泄漏

解决办法：传参context.getApplicationContext()。

* 3 属性动画导致的内存泄漏

解决办法：在Activity.onDestroy()中调用Animator.cancel()停止动画。

* 4 Handler导致的内存泄漏

解决办法：使用静态内部类+WeakReference弱引用；当外部类结束生命周期时清空消息队列。

* 5 线程导致的内存泄漏

解决办法：将AsyncTask和Runnable设为静态内部类或独立出来；在线程内部采用弱引用保存Context引用。

* 6 资源未关闭导致的内存泄漏

解决办法：在Activity销毁的时候要及时关闭或者注销。例如：  
① BraodcastReceiver：调用unregisterReceiver()注销；  
②Cursor，Stream、File：调用close()关闭；  
③Bitmap：调用recycle()释放内存（2.3版本后无需手动）。

* 7 Adapter导致的内存泄漏

详情：不使用缓存而只依靠getView() 每次重新实例化Item，会给gc制造压力。  
解决办法：在构造Adapter时使用缓存的convertView。

* 8 WebView导致的内存泄漏。

详情：WebView比较特殊，即使是调用了它的destroy方法，依然会导致内存泄漏。  
解决办法：其实避免WebView导致内存泄漏的最好方法就是让WebView所在的Activity处于另一个进程中，当这个Activity结束时杀死当前WebView所处的进程即可，我记得阿里钉钉的WebView就是另外开启的一个进程，应该也是采用这种方法避免内存泄漏。

* 9 集合类泄漏

详情：比如全局map等有静态应用，最后没有做删除。  
解决办法：在onDestry时回收不需要的集合。

* 10 扩大内存通常有两种方法：

一个是在清单文件中的Application下添加largeHeap="true"这个属性，另一个就是同一个应用开启多个进程来扩大一个应用的总内存空间。  
第二种方法其实就很常见了，比方说我使用过个推的SDK，个推的Service其实就是处在另外一个单独的进程中。

## 常见的内存泄漏场景

**1.单例造成泄漏getInstance(Context context)**

原因：单例对象一般是和应用的生命周期一样长，假如Context用activity实例化单例，会导致activity无法回收。

解决：应使用context.getApplicationContext()。

**2.非静态内部类创建静态实例造成泄漏**

原因：Java中非静态内部类持有外部类的引用，用非静态内部类创建的实例生命周期和应用一样长，导致外部类实例无法回收。

解决：内部类定义写成静态类。

**3.handler造成泄漏**

原因：message耗时任务没有处理完，message持有handler引用，handler持有activity引用，导致activity泄漏

解决：a.将handler声明为静态内部类;b.通过弱引用的方式引入activity

**4.线程造成泄漏**

原因：创建子线程asynctask和Thread/Runnable非静态内部类持有外部类引用，而子线程中耗时任务没完成activity关闭无法回收。

解决：创建子线程asynctask和Thread/Runnable定义成静态内部类

**5.webview造成泄漏**

webview打开网页会申请大量的堆内存，用于缓存网页和图片，所有webview内存泄漏应该引起重视。

原因：webview内部开起了很多线程，可能会泄漏。

解决：a.webview所在activity单独写在一个进程中，在退出activity杀死进程。

## 布局优化？

答：布局优化的本质就是减少View的层级。在Android中系统对View进行测量、布局和绘制时，都是通过对View树的遍历来进行操作的。如果一个View树 的高度太高就会严重影响测量、布局和绘制的速度。

常见的布局优化方案如下：

（1）使用RelativeLayout替代LineraLayout作为默认根布局，目的就是降低LineraLayout嵌套产生布局树的高度，从而提高UI渲染的效率。

（2）使用 < include > 标签将常用的布局组件共同的部分抽取出来，以便复用。

（3）通过 < ViewStub > 标签来加载不常用的布局，延迟加载（需要的时候在activity中加载出来）

（4）使用 < Merge > 标签来减少布局的嵌套层次

## 网络请求进行优化？

（1）尽量减少网络请求，能够合并的就尽量合并

（2）避免DNS解析，根据域名查询可能会耗费上百毫秒的时间，也可能存在DNS劫持的风险。可以根据业务需求采用增加动态更新IP的方式，或者在IP方式访问失败时切换到域名访问方式。

（3）大量数据的加载采用分页的方式

（4）网络数据传输采用GZIP压缩

（5）加入网络数据的缓存，避免频繁请求网络

（6）上传图片时，在必要的时候压缩图片

## 卡顿优化？

答：主要有：

（1）不要在主线程进行网络访问/大文件的IO操作

绘制UI时，尽量减少绘制UI层次；减少不必要的view嵌套，可以用Hierarchy Viewer工具来检测；

（2）当我们的布局是用的FrameLayout的时候，我们可以把它改成merge,可以避免自己的帧布局和系统的ContentFrameLayout帧布局重叠造成重复计算(measure和layout)

（3）提高显示速度,使用ViewStub：当加载的时候才会占用。不加载的时候就是隐藏的，仅仅占用位置。

（4）在view层级相同的情况下，尽量使用 LinerLayout而不是RelativeLayout；因为RelativeLayout在测量的时候会测量二次，而LinerLayout测量一次，可以看下它们的源码；

（5）删除控件中无用的属性;

（6）布局复用.比如listView 布局复用

（7）尽量避免过度绘制（overdraw）,比如：背景经常容易造成过度绘制。由于我们布局设置了背景，同时用到的MaterialDesign的主题会默认给一个背景。这时应该把主题添加的背景去掉；还有移除

（8）XML 中非必须的背景

（9）自定义View优化。使用 canvas.clipRect()来帮助系统识别那些可见的区域，只有在这个区域内才会被绘制。也是避免过度绘制．

（10）启动优化,启动速度的监控，发现影响启动速度的问题所在，优化启动逻辑，提高应用的启动速度。比如闪屏页面，合理优化布局，加载逻辑优化，数据准备.

合理的刷新机制，尽量减少刷新次数，尽量避免后台有高的 CPU 线程运行，缩小刷新区域。

## 启动优化？

1冷启动优化

在冷启动的时候，当应用程序进程被创建后，就需要设置启动窗口的主题。目前，大部分的 应用在启动会都会先进入一个闪屏页(LaunchActivity) 来展示应用信息，如果在Application 初始化了其它第三方的服务，就会出现启动的白屏问题。

为了更顺滑无缝衔接我们的闪屏页，可以在启动 Activity的 Theme中设置闪屏页图片，这样启动窗口的图片就会是闪屏页图片，而不是白屏。

2代码方面优化

冷启动其实只是视觉上的优化 治标不治本

adb shell am start -W packagename/[packagename].首屏Activity

答：繁琐的Application 初始化：无论问题在于不必要的初始化还是磁盘I/O，解决方案都是延迟初始化。换句话说，你应该只初始化立即需要的对象。不要创建全局静态对象，而是转向单例模式，应用程序只在第一次需要时初始化对象。  
此外，考虑使用依赖注入框架（如Hilt）

## 绘制优化

答：包括两个方面：  
（1）onDraw中不要创建新的局部对象。  
因为onDraw方法可能会被频繁调用，这样就会在一瞬间产生大量的临时对象，这不仅占用了过多的内存而且还会导致系统更加频繁gc，降低了程序的执行效率。  
（2）onDraw方法中不要做耗时的任务，  
不能执行成千上万次的循环操作，尽管每次循环都很轻量级，但是大量的循环仍然十分抢占CPU的时间片，这会造成View的绘制过程不流畅。  
按照Google官方给出的性能优化典范中的标准，View的绘制频率保证60fps是最佳的，这就要求每帧绘制时间不超过16ms(16ms = 1000/60)，虽然程序很难保证16ms这个时间，但是尽量降低onDraw方法中的复杂度总是切实有效的。

## **WebView进行优化？**

* **单/多进程化**：webView在独立的进程里面，那么WebView的进程崩溃不会影响到主进程运行；同时WebView的安全漏洞也很难影响到主进程；如果是多进程的话，可以使用WebView的容器池，有二次秒开的作用； 不过缺点就是需要你做好和WebView的跨进程通讯了
* **网络优化**：我们可以让WebView的host和客户端的host 保持一致，那么就达到复用DNS缓存的效果；如果客户端有针对网络请求进行了优化，那么可以让WebView的全部网络请求托管给客户端
* **H5离线包**：这个是手Q的H5方案之一，让客户端提前去 下载离线的H5数据包，WebView只需要加载本地H5数据包即可，这么做不仅可以避免一些http的劫持，而且跳过了WebView的建立TCP连接和H5、CCS等数据下载的过程，直接开始UI渲染，大大提高了WebView的效率

## 网页调用原生方法大概的原理

* 先让Activity注册实现schema
* Activity实现可以接收schema传过来的参数的方法
* 设置webView的WebViewClient并重写shouldOverrideUrlLoading()方法，在这个方法中实现拦截URL判断，然后传递到上序activity中进行相关处理。

## 说一说Android开发过程中性能优化所用到工具？

## Memory Monitor

## LeakCanary

## Lint

（1）Memory Monitor ： 它是Android Studio自带的一个内存监视工具，它可以很好地帮助我们进行内存实时分析。通过点击Android Studio右下角的Memory Monitor标签，打开工具可以看见较浅蓝色代表free的内存，而深色的部分代表使用的内存从内存变换的走势图变换，可以判断关于内存的使用状态，例如当内存持续增高时，可能发生内存泄漏；当内存突然减少时，可能发生GC等，如下图所示。

（2）LeakCanary： LeakCanary是Square公司基于MAT开发的一款监控Android内存泄漏的开源框架。其工作的原理是： 监测机制利用了Java的WeakReference和ReferenceQueue，通过将Activity包装到WeakReference中，被WeakReference包装过的Activity对象如果被回收，该WeakReference引用会被放到ReferenceQueue中，通过监测ReferenceQueue里面的内容就能检查到Activity是否能够被回收（在ReferenceQueue中说明可以被回收，不存在泄漏；否则，可能存在泄漏，LeakCanary是执行一遍GC，若还未在ReferenceQueue中，就会认定为泄漏）。

如果Activity被认定为泄露了，就抓取内存dump文件(Debug.dumpHprofData)；之后通过HeapAnalyzerService.runAnalysis进行分析内存文件分析；接着通过HeapAnalyzer (checkForLeak—findLeakingReference—findLeakTrace)来进行内存泄漏分析。最后通过DisplayLeakService进行内存泄漏的展示。

1. Lint ： Android Lint Tool 是Android Sutido种集成的一个Android代码提示工具，它可以给你布局、代码提供非常强大的帮助。硬编码会提示以级别警告，例如：在布局文件中写了三层冗余的LinearLayout布局、直接在TextView中写要显示的文字、字体大小使用dp而不是sp为单位，就会在编辑器右边看到提示。

# 组件化、模块化、插件化

## 组件化和模块化的区别

组件化就是把可以复用的、独立的、基础的、功能专一的代码封装到一个方法或者代码片段里，在未来需要的地方引入使用。用极少的代码实现之前相同的功能，避免了相同功能代码的复写，提高了开发的效率。在未来对改组件功能进行修改的时候只需要修改组件代码就可修改项目里所有的相同功能。组件化属于纵向分块，每个组件就像一个竖直的线永不相交。

模块化是为了单独实现某一功能模块进行封装的方法，一个模块里可能拥有n个基础组件搭配产生。模块化属于横向分块，每个模块像一条横向把n条竖直的线串联起来形成一个整体。

组件相当于库，模块相当于框架。

# Gradle、多渠道打包

# 屏幕适配

# 111

# 111