# 自我介绍

2015年毕业于北京航空航天大学北海学院，读的是计算机科学与技术专业。在大三暑期实践时接触到了Android，当时觉得能在手机上运行自己开发的应用觉得挺有趣，还可以展示给亲戚朋友看，于是我就开始往Android开发方向发展了。

平时喜欢看郭霖、洪洋这些公众号来提升自己。

2015年就职于万家游戏，从事一些Android SDK和WEB前端的一些工作。为公司开发过一个官网，长期维护公司Android SDK的广告模块。后来因为部门转为游戏开发和自己职业规划冲突而离职。

2017年在乐逗游戏做Android开发，负责更新和维护公司的渠道计费SDK，支撑公司游戏接入SDK后上线。功能包括登录、支付、退出、闪屏、分享、推送、问卷调查。每个渠道有不同sdk，每个游戏要接几十个渠道，本框架统一封装渠道接口供游戏使用，让他们只接入一个sdk，帮他们分渠道上架。并根据bugly排查崩溃问题，根据自有上报数据排查登录不上及支付优化。支付过程打点，分析游戏支付丢失问题，通过顺序可知游戏是否闪退，收到什么支付回调。为游戏解决支付掉单问题（轮询订单），为游戏playerId变化问题做udid持久化,androidId+imei/android+mac/null（imei需要动态申请权限），只保存私有空间,根据权限变化会生成不同udid，卸载重装包会导致变化，将udid保存在SDK卡可防止卸载清理私有空间数据。因为岗位没有提升空间离职。

2018年在冰川网络做android客户端H5游戏平台开发，由于团队管理与个人类型不合适没有选择转正。不开周会，需求没有交流空间，经常信息同步不到位，导致需求修改频繁。

在蜂玩做了几个区块链钱包，类似于支付宝，可用于在商店扫码购买商品，线上买卖虚拟币。

依诺唯深做了一个小视频app，期间遇到闪屏图片适配问题，当时手机同样是xhdpi的分辨率有16：9、20：9等等很多，使用drawable-long来放一张20：9的图片适配相近的比例，drawable-xhdpi里放16：9的图片来适配。布局适配框架也相似

职业规划：高级Android工程师，Android架构师。

# Android系统架构



# Okhttp

用于替代HttpUrlConnection和Apache HttpClient，android API23 6.0里已移除HttpClient。

## 原理

（1）、当我们通过OkhttpClient创建一个Call，并发起同步或异步请求时；设置连接超时之类的属性，每个OkhttpClient有独立连接池connection pool和线程池thread pools。  
（2）、okhttp会通过Dispatcher对我们所有的RealCall（Call的具体实现类）进行统一管理，并通过execute()及enqueue()方法对同步或异步请求进行处理；  
（3）、execute()及enqueue()这两个方法会最终调用RealCall中的getResponseWithInterceptorChain()方法，从拦截器链中获取返回结果；

拦截器可设置头部信息获取请求信息等。  
（4）、拦截器链中，依次通过RetryAndFollowUpInterceptor（重定向拦截器）、BridgeInterceptor（桥接拦截器）、CacheInterceptor（缓存拦截器）、ConnectInterceptor（连接拦截器）、CallServerInterceptor（网络拦截器）对请求依次处理，与服务的建立连接后，获取返回数据，再经过上述拦截器依次处理后，最后将结果返回给调用方。

## 优势

允许连接到同一个主机地址的所有请求,提高请求效率

共享Socket,减少对服务器的请求次数

通过连接池,减少了请求延迟

缓存响应数据来减少重复的网络请求

减少了对数据流量的消耗

自动处理GZip压缩

## 功能

get,post请求

文件的上传下载

加载图片(内部会图片大小自动压缩)

支持请求回调，直接返回对象、对象集合 

支持session的保持

# Volley

* 使用Volley 需要Volley.jar(120k)，加上自己的封装最多140k。
* 使用OkHttp需要 okio.jar (80k), okhttp.jar(330k)这2个jar包，总大小差不多400k,加上自己的封装，差不多得410k。

## ****Volley 的优点****

非常适合进行数据量不大，但通信频繁的网络操作

可直接在主线程调用服务端并处理返回结果

可以取消请求，容易扩展，面向接口编程

网络请求线程NetworkDispatcher默认开启了4个，可以优化，通过手机CPU数量

通过使用标准的HTTP缓存机制保持磁盘和内存响应的一致

## ****Volley 的缺点****

使用的是httpclient、HttpURLConnection

6.0不支持httpclient了，如果想支持得添加org.apache.http.legacy.jar

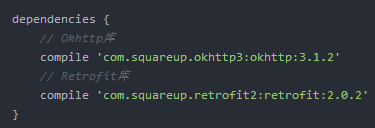
对大文件下载 Volley的表现非常糟糕

只支持http请求

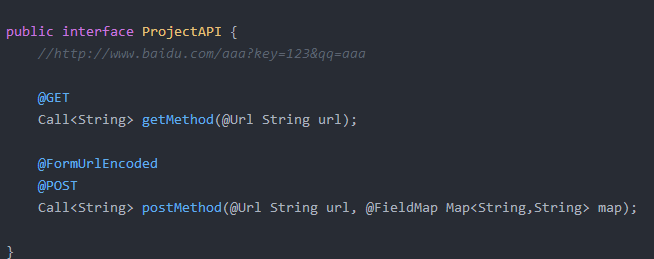
图片加载性能一般

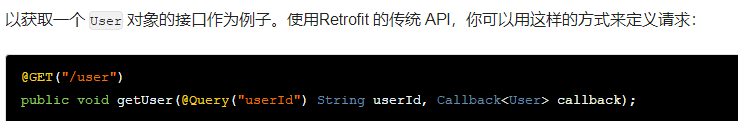
# Retrofit

是okhttp的封装，解耦，支持同步异步，支持RXJava



使用注解传入参数



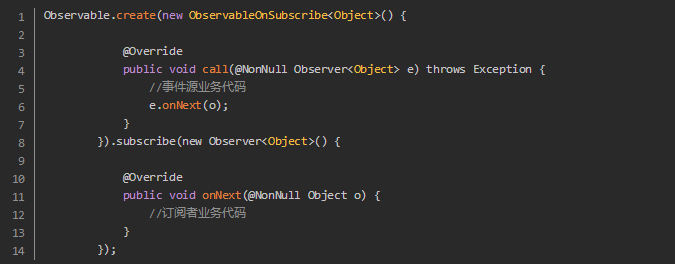


Retrofit使用动态代理模式实现我们定义的网络请求接口，在重写invoke方法的时候构建了一个ServiceMethod对象，在构建这个对象的过程中进行了方法的注解解析得到网络请求方式httpMethod，以及参数的注解分析，拼接成一个省略域名的URL

# RXJava

## 观察者

告知观察者完成某个任务，Observable（被观察者），然后创建Subscriber（观察者）



## 异步

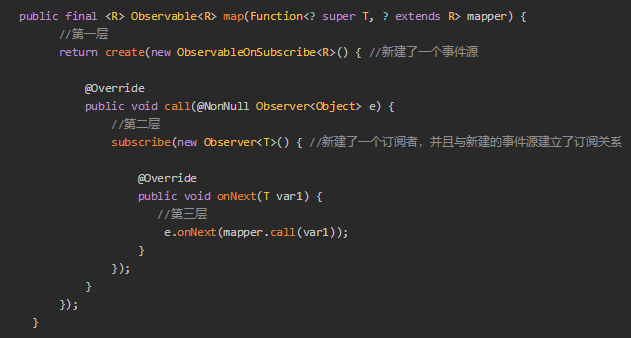
observeOn(AndroidSchedulers.mainThread())在主线程回调

subscribeOn(Schedulers.io())在子线程中执行

## map

map()的作用是将传递的数据类型进行转换，比如调用map()的上游是一个Observable<T>，经过转换以后，会将一个Observable<R>返回给调用链的下游

实现原理为在中间添加了一层订阅关系



## flatmap

链式任务用次方法，比如登录请求后支付，在此调用支付

处理返回一个被观察者Observable对象

# 动画

## 视图动画（View 动画）

动画效果：平移、旋转、缩放、透明度

1、View 动画只能为 View 添加动画效果，且不能监听 View 相关属性的变化过程。

2、View 动画提供的动画能力较为单一，目前只支持帧动画、缩放动画、位移动画、旋转动画、透明度动画以及这些动画的集合动画。

3、View动画改变的是 View 的绘制效果，View 的真正位置和相关属性并不会改变，这也就造成了点击事件的触发区域是动画前的位置而不是动画后的位置的原因。

## 帧动画

每一帧代表一个画面动作，当快速逐帧显示时，速度到达人眼无法分辨每一帧时，就达到了动画的效果。

## 属性动画

属性动画所提供的功能和 View 动画十分相似。

1、属性动画作用对象不局限在 View 上，而是任何提供了 Getter 和 Setter 方法的对象的属性上。

2、属性动画没有直接改变 View 状态的能力，而是通过动态改变 View 相关属性的方式来改变 View 的显示效果。

3、属性动画使用更方便，可以用更简洁的代码实现相关的动画效果。

4、属性动画上手难度较高，对于 propertyName 需要自己去挖掘，或者自己通过 Wrapper 的方式去自定义 propertyName。

5、属性动画是 Android3.0 以上系统提供的能力，在 3.0 以下需导入 nineoldandroids 三方库解决兼容性问题。

## 触摸反馈动画（Ripple Effect）

所谓触摸反馈动画就是一种点击效果，作用在可点击的 View 上时，当有点击事件时会有涟漪般的反馈效果，使用在 按钮 上是再好不过了。

## 揭露动画（Reveal Effect）

揭露动画在系统中很常见，就是类似波纹的效果， 从某一个点向四周展开或者从四周向某一点聚合起来。

1、可以用在 Activity 里面的 View 动画效果，用来揭露某个隐藏 View 的显示；

2、也可以使用在 Activity 跳转过渡动画中。

## Activity 切换动画

1、转场效果我们一般用在 Activity 切换时的动画效果上；

2、共享元素一般我们使用在转换的前后两个页面有共同元素[注1]时；

3、同时也可以在 Activity 布局发生场景变化时，让其中的 View 产生相应的过渡动画。

## 矢量图动画（Vector 动画）

## 视图状态动画（Animate View State Changes）

所谓视图状态动画，就是 View 在状态改变时执行的动画效果。和之前我们通过 selector 选择器给 Button 设置不同状态下的背景效果是一样一样的。

# 图片优化

三级缓存

内存缓存，读取速度最快。

硬盘缓存（文件缓存），读取速度比内存缓存稍慢。

网络缓存，读取速度最慢。

使用LruCache来缓存图片，算法就是当缓存空间满了的时候，将最近最少使用的数据从缓存空间中删除以增加可用的缓存空间来缓存新数据。将Bitmap的软引用存入Map中。

使用DiskLruCache来保存在SD卡里，文件IO流

# 反射

JAVA反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为java语言的反射机制。

类型的对象，而我不知道你具体是什么类，用这种方法

Person p1 = new Person();

Class c1 = p1.getClass();

直接通过 类名.class 的方式得到,该方法最为安全可靠，程序性能更高Class c2 = Person.class;

Class c3 = Class.forName("com.ys.reflex.Person");

# ****Map、Set、List****

## 1 HashMap、ConcurrentHashMap、HashTable

HashMap是支持null键和null值的，而HashTable在遇到null时，会抛出NullPointerException异常。

HashTable是同步的，HashMap不是。

ConcurrentHashMap采用锁分段技术，将整个Hash桶进行了分段segment，也就是将这个大的数组分成了几个小的片段segment，而且每个小的片段segment上面都有锁存在，那么在插入元素的时候就需要先找到应该插入到哪一个片段segment，然后再在这个片段上面进行插入，而且这里还需要获取segment锁。

## 2 ArrayList、LinkedList

ArrayList是实现了基于动态数组的数据结构。

LinkedList基于链表的数据结构。

对于随机访问get和set，ArrayList优，因为LinkedList要移动指针。

对于新增和删除操作add和remove，LinkedList比较占优势，因为ArrayList要移动数据。

## 3 HashSet、TreeSet

HashSet是使用散列表进行存储，元素无序，元素允许为null。TreeSet是使用树结构来进行存储，元素按字符串顺序排序存储，元素不允许为null。

# 多进程通讯

AIDL：功能强大，支持进程间一对多的实时并发通信，并可实现 RPC (远程过程调用)。服务同时处理多个请求。

Messenger：支持一对多的串行实时通信， AIDL 的简化版本。会在单一线程中创建包含所有客户端请求的队列,以便服务一次接收一个请求。AIDL的简化版本，是AIDL的封装。

Bundle：四大组件的进程通信方式，只能传输 Bundle 支持的数据类型。

ContentProvider：强大的数据源访问支持，主要支持 CRUD 操作，一对多的进程间数据共享，例如我们的应用访问系统的通讯录数据。

BroadcastReceiver：即广播，但只能单向通信，接收者只能被动的接收消息。

文件共享：在非高并发情况下共享简单的数据。

Socket：通过网络传输数据。

# ****线程****

## 1创建线程的方式

多进程是指操作系统能同时运行多个任务（程序）。

多线程是指在同一程序中有多个顺序流在执行。

通过Runnable接口创建线程类，Runnable是接口，可以再继承其他类。

继承Thread类创建线程

使用Callable和Future创建线程

## 2Join、yeild、sleep、wait、notify、notifyall

sleep()方法不会释放对象锁，一段时间后会继续线程。

wait()方法会放弃对象锁，线程进入等待状态，会等待notify()调用才会继续线程。

## 3线程池

如果并发的线程数量很多，并且每个线程都是执行一个时间很短的任务就结束了，这样频繁创建线程就会大大降低系统的效率，因为频繁创建线程和销毁线程需要时间。根据获取cpu的核心来开线程。

使用单例线程池

// 获取单例的线程池对象

public static ThreadPool getThreadPool() {

if (mThreadPool == null) {

synchronized (ThreadManager.class) {

if (mThreadPool == null) {

int cpuNum = Runtime.getRuntime().availableProcessors();// 获取处理器数量

int threadNum = cpuNum \* 2 + 1;// 根据cpu数量,计算出合理的线程并发数

mThreadPool = new ThreadPool(threadNum, threadNum, 0L);

}

}

}

return mThreadPool;

}

// 获取单例的线程池对象

public static ThreadPool getThreadPool() {

if (mThreadPool == null) {

synchronized (ThreadManager.class) {

if (mThreadPool == null) {

int cpuNum = Runtime.getRuntime().availableProcessors();// 获取处理器数量

int threadNum = cpuNum \* 2 + 1;// 根据cpu数量,计算出合理的线程并发数

mThreadPool = new ThreadPool(threadNum, threadNum, 0L);

}

}

}

return mThreadPool;

}

## 4多线程原理

同一时间内，CPU只能处理1条线程，只有1条线程在工作（执行）;多线程并发（同时）执行，其实是CPU快速地在多条线程之间调度（切换）。如果CPU调度线程的时间足够快，就造成了多线程并发执行的假象。

## 4死锁

线程死锁是指由于两个以上线程互相持有对方所需要的资源，导致这些线程处于等待状态，无法前往执行。将锁释放可解决，比如调用wait()。

1. 互斥条件：进程对于所分配到的资源具有排它性，即一个资源只能被一个进程占用，直到被该进程释放
2. 请求和保持条件：一个进程因请求被占用资源而发生阻塞时，对已获得的资源保持不放。
3. 不剥夺条件：任何一个资源在没被该进程释放之前，任何其他进程都无法对他剥夺占用
4. 循环等待条件：当发生死锁时，所等待的进程必定会形成一个环路（类似于死循环），造成永久阻塞。

# ****String****

方式一：String a = “aaa” ;

方式二：String b = new String(“aaa”);

两种方式都能创建字符串对象，但方式一要比方式二更优。

因为字符串是保存在常量池中的，而通过new创建的对象会存放在堆内存中。

# ****基本数据类型****

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 简单类型 | boolean | byte | char | short | Int | float | long | double | void |
| 二进制位数 | 1 | 8 | 16 | 16 | 32 | 32 | 64 | 64 | -- |
| 封装器类 | Boolean | Byte | Character | Short | Integer | Float | Long | Double | Void |

# ****http和https****

HTTPS：是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。

HTTPS和HTTP的区别主要如下：

　　1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

　　2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

　　3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

https：1客户端向服务端获取服务端公钥和证书，向ca验证证书。2客户端随机生成对称加密密钥，用公钥加密传给服务端。3服务端用密钥解密获取客户端密钥，用客户端密钥加密传输信息给客户端。

客户端密钥对称加密，即用于服务端加密，也用于客户端解密。

服务端密钥非对称加密，公钥用于客户端加密，私钥用于服务端解密。

# ****tcp和udp****

TCP，面向连接，可靠，传输大量数据时使用，速度慢。

是基于连接的协议，在正式收发数据前，必须和对方建立可靠的连接。三次握手：客户端向服务端确认连接（SYN），服务器向客户端确认连接（SYN+ACK），客户端确认服务端已确认收到。

四次挥手：客户端向服务端确认关闭（FIN），服务器向客户端确认收到（FIN +ACK），服务器向客户端确认关闭（FIN），客户端向服务器确认收到（FIN +ACK）。

UDP，面向非连接，不可靠，传输少量数据时使用，速度快

它是面向非连接的协议，它不与对方建立连接，而是直接就把数据包发送过去



# ****接口和抽象类****



接口成员变量默认为public static final，必须赋初值，不能被修改。

抽象类既可以有变量，也可以有常量。abstract不能修饰成员变量。

抽象类只是普通类多了抽象方法。

# ****设计模式****

## ****单例模式****

1没有延迟加载，无线程问题，但是如果未使用会造成内存浪费。

public class Singleton {

private final static Singleton INSTANCE = new Singleton();

private Singleton(){}

public static Singleton getInstance(){

return INSTANCE;

}

}

2单线程中使用，线程不安全。

public class Single{

private static Single instance;

private Single(){}

public static Single getInstance(){

if(instance==null){

instance = new Single();

}

return instance;

}

}

3线程安全，保证在不同线程中不能同时实例化对象。效率低，每次都需要同步。

public class Single{

private static Single instance;

private Single(){}

public static synchronizedSingle getInstance(){

if(instance==null){

instance = new Single();

}

return instance;

}

}

4 3的优化，提高了效率，但是线程不安全，

public class Singleton {

private static Singleton singleton;

private Singleton() {}

public static Singleton getInstance() {

if (singleton == null) {

synchronized (Singleton.class) {

singleton = new Singleton();

}

}

return singleton;

}

}

5双重锁，线程安全，延迟加载，效率高。

public class Singleton {

private static volatile Singleton singleton;

private Singleton() {}

public static Singleton getInstance() {

if (singleton == null) {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

}

return singleton;

}

}

6 线程安全，延迟加载，效率高。

public class Singleton {

private Singleton() {}

private static class SingletonInstance {

private static final Singleton INSTANCE = new Singleton();

}

public static Singleton getInstance() {

return SingletonInstance.INSTANCE;

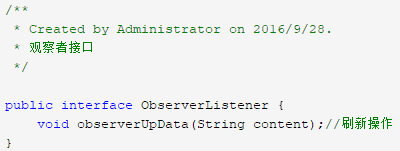
}

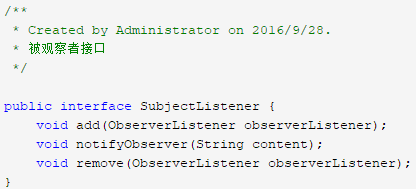
}

## 观察者模式

观察者模式定义了一个一对多的依赖关系，让多个观察者对象同时监听一个主体对象，当这个主体对象改变时，会通知所有的观察者，让他们能够自动的更新自己。

类似广播，订阅之后，数据变化时给予回调。







## Builer模式

Builder模式是一步一步创建一个复杂对象的创建型模式，它允许用户在不知道内部构建细节的情况下，可以更精细的控制对象的构造流程。该模式是为了将构建复杂对象的过程和它的部件解耦，使得构建过程和部件的表示隔离开来，两者之间的耦合度也降到最低。

调用setxx()一直返回这个对象this

## 代理模式

代理(Proxy)是一种设计模式,提供了间接对目标对象进行访问的方式;即通过代理对象访问目标对象.这样做的好处是:可以在目标对象实现的功能上，增加额外的功能补充，即扩展目标对象的功能。

这就符合了设计模式的开闭原则，即在对既有代码不改动的情况下进行功能的扩展。

静态代理：接口A方法a，类B1、B2、B3实现A方法a，类C实现A并构造传入一个A类型b1、b2、b3，C.a()调用传入对象的方法a，b1时调用b1.a()，b2时调用b2.a()。

动态代理：反射调用传入类的方法



## 适配器模式

Adapter(适配器接口):即目标角色，定义把其他类转换为何种接口，也就是我们期望的接口。

Adaptee(被适配角色):即源角色，一般是已存在的类，需要适配新的接口。

ConcreteAdapter(具体适配器):实现适配器接口，把源角色接口转换为目标角色期望的接口。

C实现接口A继承类B，作为B的子类多出了A的方法。开闭原则

# 卡顿

UI线程无法按时刷新，导致掉帧，便是卡顿。卡顿导致ANR。主线程耗时操作（网络请求、文件读写、数据库、大数据量的sp），主线程等待事件完成，内存泄漏导致的频繁GC

UI线程是基于queue中的message事件驱动的，事件 -> 执行 -> 下一个事件，另一方面由于Android的帧率是60fps，也就是每16ms就会触发一次UI刷新，如果某个message的处理时间 > 16ms，就会导致接收到VSYNC信号的时候无法完成本次刷新操作，产生掉帧现象。

因此，从本质上来讲，我们必须让UI线程的任何事件在16ms之内解决战斗。

基于此，可能会导致卡顿的原因有三大类：

1. UI造成的卡顿：复杂布局，过度绘制，动画
2. 主线程耗时操作
3. 频繁GC

## UI卡顿

减少布局的嵌套

炫酷的动画根据机型判断是否使用

## 耗时等待

主线程耗时操作：网络请求、文件读写、数据库、大数据量的sp

## CPU时间片

1、其他应用发生抢占CPU资源的情况，导致本应用无法获得CPU执行时间片。

2、线程间发生死锁，UI线程无法获取锁，导致无法继续执行。

3、频繁GC，内存抖动。大量创建又很快销毁对象。GC的次数越多，消耗在GC上的时间就越长，CPU花在界面绘制上的时间相应就越短。

## 内存泄漏

内存泄漏导致内存不够用，频繁GC，导致卡顿，可能ANR。

# ANR

Activity 5秒内无法响应用户输入事件(例如键盘输入, 触摸屏幕等)

BroadcastReceiver在10秒内无法结束。

在主线程(UI线程)里面做了太多的阻塞耗时操作，例如文件读写，数据库读写，网络请求，图片加载

ANR产生时，系统会生成一个traces.txt的文件放在/data/anr/下。 可以通过[adb](https://www.jianshu.com/p/5980c8c282ef" \t "_blank)命令将其导出到本地：

adb pull /data/anr .

分析traces.txt文件中的日志有哪些耗时操作，有说明原因

# 判断主线程

方法一：使用 Looper 类判断   
Looper.myLooper() != Looper.getMainLooper()   
方法二：通过查看 Thread 类的当前线程   
Thread.currentThread() == Looper.getMainLooper().getThread()   
方法三：打印 Log，去看线程 id，看是否和进程号一样，一样是主线 程

# Singletop和SingleTask的不同

SingleTop 栈顶复用模式

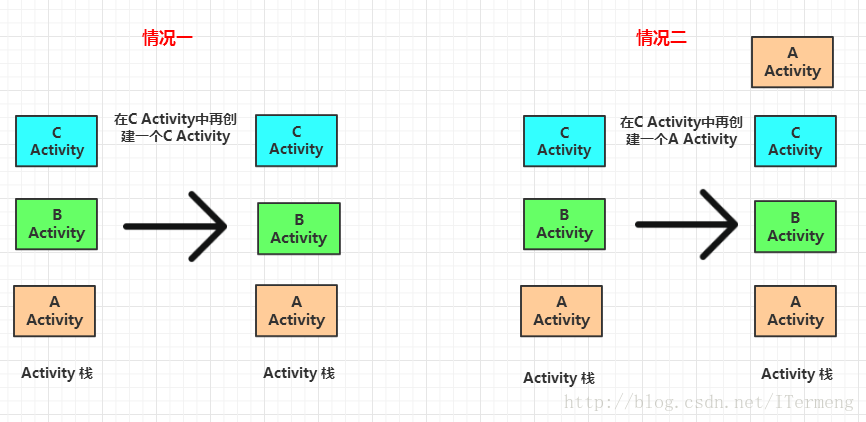
说明：分两种处理情况：须要创建的Activity已经处于栈顶时，此时会直接复用栈顶的Activity。不会再创建新的Activity；若须要创建的Activity不处于栈顶，此时会又一次创建一个新的Activity入栈，同Standard模式一样。

生命周期：若情况一中栈顶的Activity被直接复用时，它的onCreate、onStart不会被系统调用，由于它并没有发生改变。可是一个新的方法 onNewIntent会被回调（Activity被正常创建时不会回调此方法）。

举例：此时Activity 栈中以此有A、B、C三个Activity，此时C处于栈顶，启动模式为SingleTop 模式。

情况一：在C Activity中加入点击事件，须要跳转到还有一个同类型的C Activity。结果是直接复用栈顶的C Activity。

情况二：在C Activity中加入点击事件，须要跳转到还有一个A Activity。结果是创建一个新的Activity入栈。成为栈顶。



SingleTask 栈内复用模式

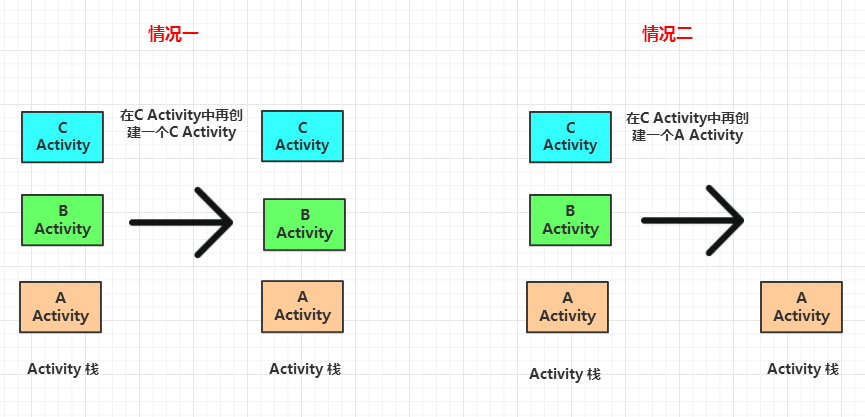
说明：若须要创建的Activity已经处于栈中时，此时不会创建新的Activity，而是将存在栈中的Activity上面的其他Activity所有销毁，使它成为栈顶。

生命周期：同SingleTop 模式中的情况一同样。仅仅会又一次回调Activity中的 onNewIntent方法

举例：此时Activity 栈中以此有A、B、C三个Activity。此时C处于栈顶，启动模式为SingleTask 模式。

情况一：在C Activity中加入点击事件，须要跳转到还有一个同类型的C Activity。结果是直接用栈顶的C Activity。

情况二：在C Activity中加入点击事件，须要跳转到还有一个A Activity。结果是将A Activity上面的B、C所有销毁，使A Activity成为栈顶。

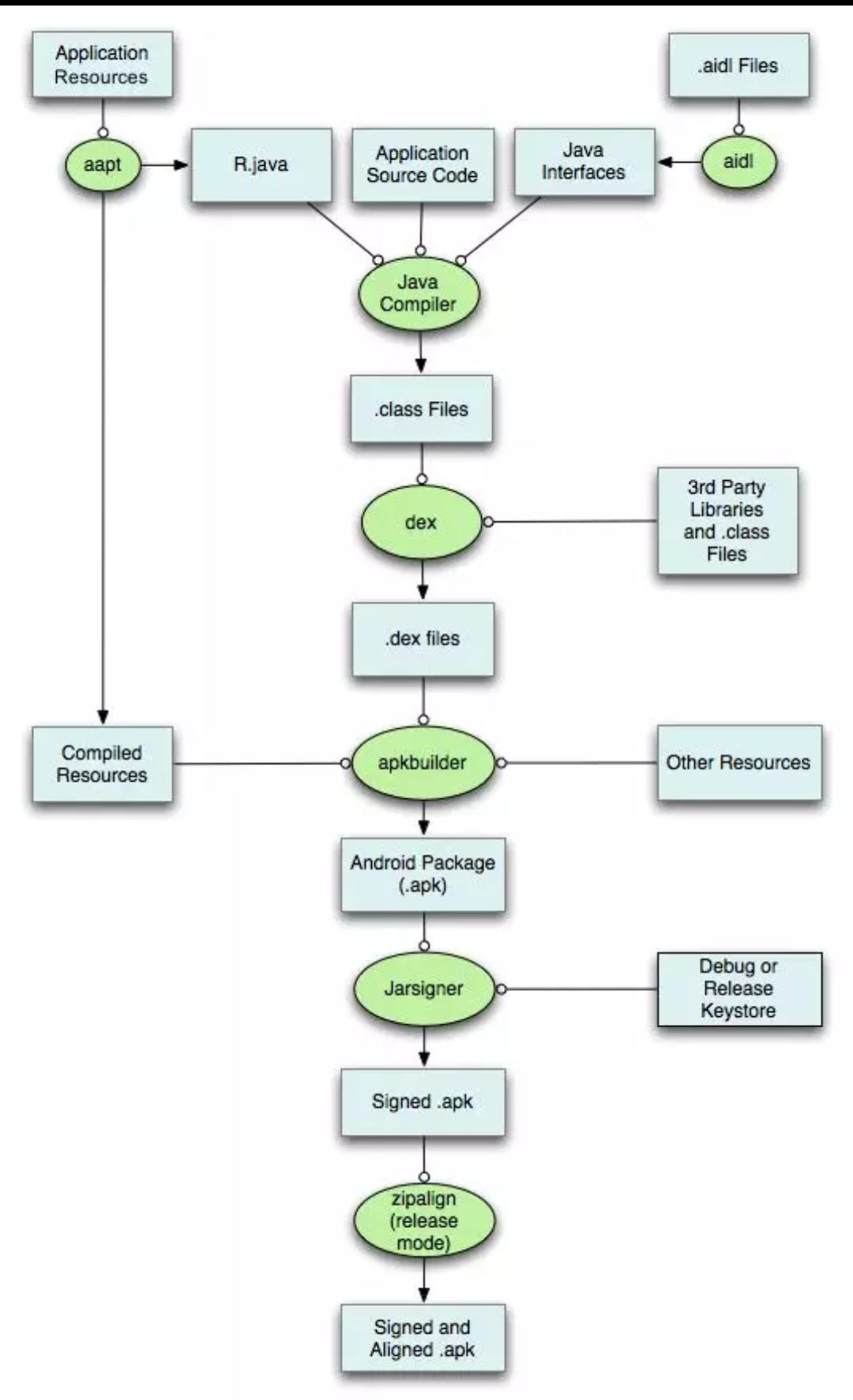


# APK反编译流程

Apktool反编译apk成samli文件，dex2jar将classes.dex编译成jar，jd-gui查看源码

# APK编译流程

1. 工程的资源文件(res文件夹下的文件)，通过AAPT打包成R.java类(资源索引表)，以及.arsc资源文件 。Manifest被编译成二进制xml，arsc是资源索引，R.java是资源常量池。
2. 如果有aidl（跨进程通信），通过aidl工具，打包成java接口类
3. R.java和aidl.java通过java编译成想要的.class文件。
4. 源码class文件和第三方jar或者library通过dx工具打包成dex文件。dx工具的主要作用是将java字节码转换成Dalvik字节码，在此过程中会压缩常量池，消除一些冗余信息等。
5. apkbuilder工具会将所有没有编译的资源，.arsc资源，.dex文件打包到一个完成apk文件中。
6. 签名，5中完成apk通过配置的签名文件(debug和release都有)，jarsigner工具会对齐签名。得到一个签名后的apk,signed.apk
7. zipAlign工具对6中的signed.apk进行对齐处理，所谓对齐，主要过程是将APK包中所有的资源文件距离文件起始偏移为4字节整数倍，这样通过内存映射访问apk文件时的速度会更快。对齐的作用主要是为了减少运行时内存的使用。



# AIDL

设计这门语言的目的是为了实现进程间通信，尤其是在涉及多进程并发情况下的进程间通信。

每一个进程都有自己的Dalvik VM实例，都有自己的一块独立的内存，都在自己的内存上存储自己的数据，执行着自己的操作，都在自己的那片狭小的空间里过完自己的一生。每个进程之间都你不知我，我不知你，就像是隔江相望的两座小岛一样，都在同一个世界里，但又各自有着自己的世界。而AIDL，就是两座小岛之间沟通的桥梁。相对于它们而言，我们就好像造物主一样，我们可以通过AIDL来制定一些规则，规定它们能进行哪些交流——比如，它们可以在我们制定的规则下传输一些特定规格的数据。

总之，通过这门语言，我们可以愉快的在一个进程访问另一个进程的数据，甚至调用它的一些方法，当然，只能是特定的方法。

但是，如果仅仅是要进行跨进程通信的话，其实我们还有其他的一些选择，比如 BroadcastReceiver , Messenger 等，但是 BroadcastReceiver 占用的系统资源比较多，如果是频繁的跨进程通信的话显然是不可取的；Messenger 进行跨进程通信时请求队列是同步进行的，无法并发执行。

# Context

Context一共有三种类型，分别是Application、Activity和Service。

Activity的父类是ContextThemeWrapper，需要带主题。

有些控件可使用无主题的Context，有些一定要主题。

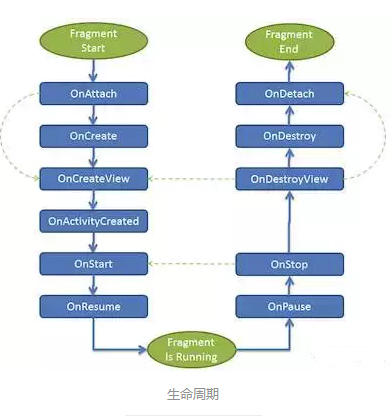
Context数量 = Activity数量 + Service数量 + 1

# Activity

1. onCreate：与onDestroy配对，表示Activity正在被创建，这是生命周期的第一个方法。在这个方法中可以做一些初始化的工作（加载布局资源、初始化Activity所需要的数据等），耗时的工作在异步线程上完成。
2. onRestart：表示Activity正在重新启动。一般情况下，在当前Activity从不可见重新变为可见的状态时onRestart就会被调用。这种情形一般是由于用户的行为所导致的，比如用户按下Home键切换到桌面或者打开了一个新的Activity（这时当前Activity会暂停，也就是onPause和onStop被执行），接着用户有回到了这个Activity，就会出现这种情况。
3. onStart：与onStop配对，表示Activity正在被启动，并且即将开始。但是这个时候要注意它与onResume的区别。两者都表示Activity可见，但是onStart时Activity还正在加载其他内容，正在向我们展示，用户还无法看到，即无法交互。
4. onResume：与onPause配对，表示Activity已经创建完成，并且可以开始活动了，这个时候用户已经可以看到界面了，并且即将与用户交互（完成该周期之后便可以响应用户的交互事件了）。
5. onPause：与onResume配对，表示Activity正在暂停，正常情况下，onStop接着就会被调用。在特殊情况下，如果这个时候用户快速地再回到当前的Activity,那么onResume会被调用（极端情况）。一般来说，在这个生命周期状态下，可以做一些存储数据、停止动画的工作，但是不能太耗时，如果是由于启动新的Activity而唤醒的该状态，那会影响到新Activity的显示，原因是onPause必须执行完，新的Activity的onResume才会执行。
6. onStop：与onStart配对，表示Activity即将停止，可以做一些稍微重量级的回收工作，同样也不能太耗时（可以比onPause稍微好一点）。
7. onDestroy：与onCreate配对，表示Activity即将被销毁，这是Activity生命周期的最后一个回调，我们可以做一些回收工作和最终的资源释放（如Service、BroadReceiver、Map等）。



# Fragment

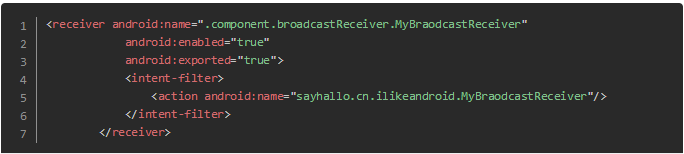


onAttach() 在Fragment 和 Activity 建立关联是调用（Activity 传递到此方法内）  
onCreateView() 当Fragment 创建视图时调用  
onActivityCreated() 在相关联的 Activity 的 onCreate() 方法已返回时调用。  
onDestroyView() 当Fragment中的视图被移除时调用  
onDetach() 当Fragment 和 Activity 取消关联时调用。

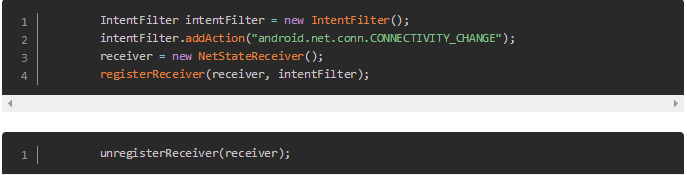
# BroadcastReceiver

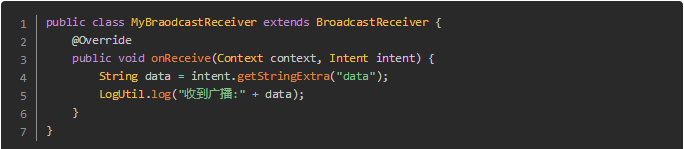
有全局广播，有序广播，本地广播

静态注册：在AndroidManifest.xml 中，通过标签来声明



优点：不受其他组件生命周期影响，即使应用程序被关闭，也能接收广播。会在接收到广播之后销毁  
缺点：耗电，占内存。  
适用场景：需要时刻监听的广播

动态注册：在代码中注册



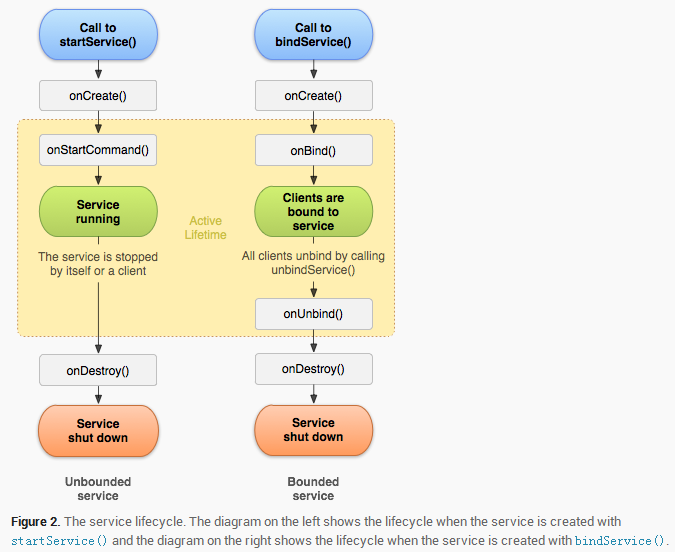
优点：灵活，不耗电，易控，省内存  
缺点：需要手动注销  
适用场景：需要特定时候监听的广播

# Service

使用startService()方法启用服务，调用者与服务之间没有关连，即使调用者退出了，服务仍然运行。stopService()。

当我们通过startService被调用以后，多次在调用startService(),onCreate()方法也只会被调用一次，而onStartConmon()会被多次调用当我们调用stopService()的时候，onDestroy()就会被调用，从而销毁服务

使用bindService()方法启用服务，调用者与服务绑定在了一起，调用者一旦退出，服务也就终止。unbindService()。



# SharedPreference

根据sp的设计，创建的时候会开启子线程把整个文件全部加载进内存，加载完毕再通知主线程，如果读取尚未结束，此时想获取某个key的值，主线程就必须等待加载完毕为止。

因此，如果sp文件比较大，那么会带来几个严重问题：

1第一次从sp中获取值的时候，有可能阻塞主线程，使界面卡顿、掉帧。

2解析sp的时候会产生大量的临时对象，导致频繁GC，引起界面卡顿。

3这些key和value会永远存在于内存之中，不会被释放，占用大量内存。

所以千万不要把庞大的key/value存在sp中，比如把复杂的json当value。

另外对于sp的存储，commit是同步操作，要在子线程中使用。而apply虽然是在子线程执行的，但是无节制地apply也会造成卡顿，原因是每次有系统消息发生的时候（handleStopActivity，handlePauseActivity）都会去检查已经提交的apply写操作是否完成，如果没有完成则阻塞主线程。

# IntentService

1、IntentService 是继承自 Service 并处理异步请求的一个类，在 IntentService 内有一个工作线程来处理耗时操作。

2、当任务执行完后，IntentService 会自动停止，不需要我们去手动结束。

3、如果启动 IntentService 多次，那么每一个耗时操作会以工作队列的方式在 IntentService 的 onHandleIntent 回调方法中执行，依次去执行，使用串行的方式，执行完自动结束。

# Listview优化

1. dapter的getView(int position, ViewconvertView, ViewGroup parent)方法中，第二个参数convertView代表的就是之前滑动出屏幕的条目对象。如果是第一次加载该方法时，会创建新的View对象，如果滑动ListView时，滑动出屏幕的View对象会以缓存的形式存在，而convertView就是缓存的View对象，我们可以复用缓存该对象减少新对象的创建。在加载布局时先判断convertView是否存在，如果convertView==null说明没有缓存的View对象，则使用View.inflate()方法加载布局，进行布局的初始化，否则复用缓存的View对象。
2. findviewById重新找到控件，然后对控件进行赋值，这样会减慢加载的速度，可以创建一个内部类ViewHolder，里面的成员变量和view中所包含的组件个数、类型相同，在convertview为null的时候，把findviewbyId找到的控件赋给ViewHolder中对应的变量，就相当于先把它们装进一个容器，下次要用的时候，直接从容器中获取不需要再findviewById
3. 有些情况下我们需要加载网络中的数据，显示到ListView，而往往此时都是数据量比较多的一种情况，如果数据有1000条，没有优化过的ListView都是会一次性把数据全部加载出来的，很显然需要一段时间才能加载出来，我们不可能让用户面对着空白的屏幕等好几分钟，那么这时我们可以使用分段加载，比如先设置每次加载数据10条，当用户滑动ListView到底部的时候，我们再加载20条数据出来，然后使用Adapter刷新ListView，这样用户只需要等待10条数据的加载时间，这样也可以缓解一次性加载大量数据而导致OOM崩溃的情况。
4. 上面第三种方式其实也不能完全解决OOM崩溃的情况，因为虽然我们在分段中一次只增加10条数据到List集合中，然后再刷新到ListView中去，假如有10万条数据，如果我们顺利读到最后这个List集合中还是会累积海量条数的数据，还是可能会造成OOM崩溃的情况，这时候我们就需要用到分页，比如说我们将这10万条数据分为1000页，每一页100条数据，每一页加载时都覆盖掉上一页中List集合中的内容，然后每一页内再使用分批加载，这样用户的体验就会相对好一些。

# oom

使用AS的monitors导出hporf文件（dump Java heap按钮），用sdk\platform-tools\ hprof-conv.exe转换成eclipse格式的hporf文件（hprof-conv n:\1.hprof n:\2.hprof），然后用eclipse的mat分析。。。

MAT集成方法：1.Help->Eclipse Marketplace->查找Memory Analysis->install->重启eclipse-> Window-> open perspective->打开Memory Analysis

充分挖掘特征。在挖掘特征时，需要多方面考虑，此过程更多的是猜测怀疑，所以可能的方面都要考虑到，包括但不限于代码改动、机器特征、时间特征等，必要时还需要做一定的统计分析。

根据掌握的特征寻找稳定的复现的途径。一般需要做内存压力测试，这样比较容易达到OOM的临界值，只是简单的一些正常操作难以触发OOM。

获取可分析的数据（内存dump文件）。利用MAT分析dump文件，MAT可以方便的按照大小排序实例，可以查看某些实例到GC ROOT的路径。

# AsyncTask

Android封装的异步线程类，AsyncTask的内部Handler和ThreadPoolExecutor都是进程范围内共享

# 版本适配

targetSdkVersion

## 6.0

动态申请权限

移除Appache的HTTP Client

## 7.0

应用间共享文件URI不能使用file://需要使用 content://，manifest文件中注册FileProvider

## 8.0

Android8.0中androidID算法和之前不一样，应用签署密钥、用户和设备的每个组合都具有唯一的 androidID 值

Icon，会默认创建圆形图标。可定制icon形状

# handler消息机制

应用会把操作转换成消息让主线程处理，应用启动的时候会通过ActivityThread创建一个主线程，在主线程中关联一个Looper不断从消息队列MessageQueue中获取消息，消息队列对应了一个handler传送消息到消息队列中。

通过要传送的消息保存到Message中，handler通过调用sendMessage方法将Message发送到MessageQueue中，Looper对象就会不断的调用loop()方法不断的从MessageQueue中取出Message交给handler进行处理。从而实现线程之间的通信。

避免内存泄漏： Handler持有外部Acvitity对象，如果handler发送的消息在当前handler的消息队列中，但acvitity finsh则泄漏。创建静态内部类继承Handler。

# 强更

1. 检查本地版本是否最新版本
2. dialog提示用户更新
3. dialog显示下载进度
4. 下载最新版本APK
5. 下载完成提示安装
6. Intent startActivity安装

静默安装：需要root权限，用pm install指令安装APK



# 静态内部类和非静态内部类

内部类是外部类的方法。

静态内部类可以有静态成员（方法和属性），而非静态内部类则不能有静态成员（成员或属性）

静态内部类只能够访问外部类的静态成员和静态方法，而非静态内部类则可以访问外部类的所有成员（方法和属性）

实例化一个非静态的内部类的方法：

OutClass.InnerClass  innerClass = new OutClass().new InnerClass();

实例化一个静态内部类的方法：

不依赖于外部类的实例，直接实例化静态内部类对象：

OutClass.InnerClass innerClass =  new OutClass.InnerClass();

内部类相对于外部类来说，前者是后者的方法，所以当前者是static修饰的时候，在前者中不能访问后者中的非static属性和方法；相反，当前者是非static修饰的时候，前者可以访问后者中的任何成员。

什么时候用静态内部类？什么时候用非静态内部类？

1）首先要弄清楚：为什么要用内部类？用内部类是因为内部类与所在外部类有一定的关系，往往只有该外部类调用此内部类，

所以没有必要专门用一个Java文件存放这个类。

2）生命周期不一样：静态内部类随着外部类的加载而加载，而不是随着外部类对象的产生而产生。

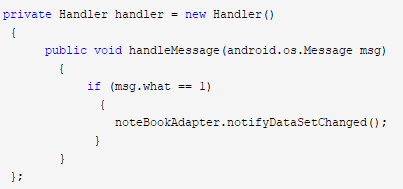
外部类实例与静态内部类实例是没有关系的。

外部内部类实例对应着不同的非静态内部类实例。

# 避免内存泄漏

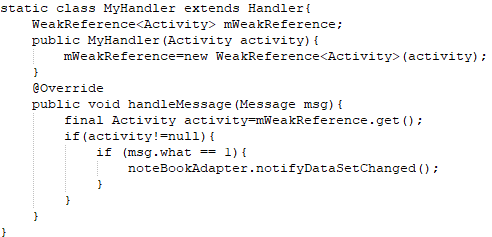
1. activity存活在它的生命周期之外。在类中定义了静态Activity变量，把当前运行的Activity实例赋值于这个静态变量。单例持有了activity。
2. 资源未关闭造成的内存泄漏在使用完BraodcastReceiver，ContentObserver，File，Cursor，Stream，Bitmap等资源时，一定要在Activity中的OnDestry中及时的关闭、注销或者释放内存，否则这些资源不会被GC回收，就会造成内存泄漏。
3. 非静态内部类创建的静态实例造成的内存泄漏。在Java 中，非静态的内部类和匿名内部类都会隐式地持有其外部类的引用，将内部类改为静态内部类。
4. AsynTask、Handler、Thread ，activity生命周期结束但是耗时任务为执行完。需要创建静态内部类来继承，并使用WeakReference<Activity>来弱引用activity判断是否执行。

当使用内部类（包括匿名类）来创建Handler的时候，Handler对象会隐式地持有一个外部类对象（通常是一个Activity）的引用。



可将Handler声明为静态类。静态的内部类不会持有外部类的引用，所以Activity可以随意被回收。

由于Handler不再持有外部类对象的引用，导致程序不允许在Handler中操作Activity中的对象了。所以需要在Handler中增加一个对Activity的弱引用（WeakReference）。



# EventBus

EventBus是一种用于Android的发布/订阅事件总线。

它有很多优点：简化应用组件间的通信；解耦事件的发送者和接收者；避免复杂和容易出错的依赖和生命周期的问题；很快，专门为高性能优化过等等。

粘性事件：在发送事件之后再订阅该事件也能收到该事件。

compile 'org.greenrobot:eventbus:3.1.1'

// 注册订阅者

EventBus.getDefault().register(this);

// 注销订阅者

EventBus.getDefault().unregister(this);

// 注解定义接受消息的方法。优先级数值越大，优先级越高。优先级只有在相同的线程模式下才有效。

@Subscribe(threadMode = ThreadMode.MAIN处理线程,priority = 1优先级,sticky = true粘性事件)

public void xx(Object event) {

...

// 取消事件

EventBus.getDefault().cancelEventDelivery(event);

// 移除粘性事件

EventBus.getDefault().removeStickyEvent(event);

}

// 发送消息

EventBus.getDefault().post(obj);

// 发布粘性事件

EventBus.getDefault().postSticky(obj);

## 线程切换原理

获取主线程的Looper对象并存起来。

发送消息时当前线程与主线程Looper对象做比较，如果相同，说明是在主线程，如果不同，说明是在子线程。

接受消息时需要切换线程判断是否在主线程，不是则使用handler机制传输msg，需要子线程则使用线程池避免开线程的消耗

# 自定义View

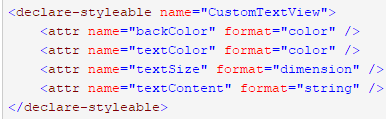
View的绘制基本上由onMeasure()测量宽高、onLayout()确定位置、onDraw()绘制、onTouchEvent()点击事件这几个函数完成

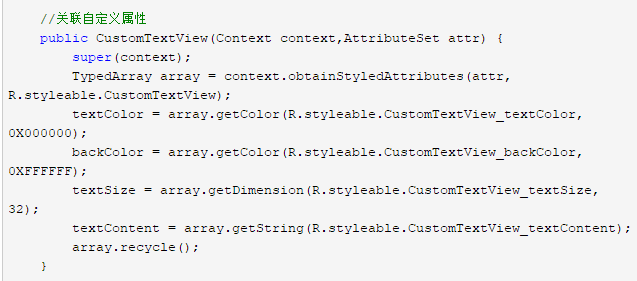
1继承View

2在res/values目录下建立attrs.xml文件，写下需要定义的属性

3编写构造函数

4重写onDraw绘制控件



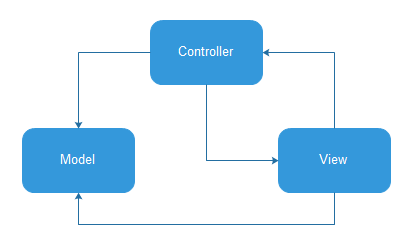




 从顺序来看，onLayout动作和onMeasure一样，都是从小往上层调用。从叶子child的onLayout调用开始，最后调用到root的onLayout方法。因为只有大小发生了变化才会执行onSizeChange,所以没有onSizeChange的时候，就是从尾到头执行完onMeasure，再开始从尾到头执行完onLayout

4. 由此看来，onSizeChange并不一定会调用，只有View的大小发生变化才会调用，而且也不一定一定从root开始调用。onMeasure在整个界面上需要放置一样东西或拿掉一样东西时会调用。比如addView就是放置，removeview就是拿掉，另外比较特殊的是，child设置为gone会触发onMeasure，但是invisible不会触发onMeasure。一旦执行过onMeasure，往往就会执行onLayout来重新布局

# MVC

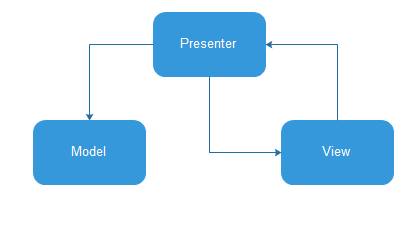


其中View层其实就是程序的UI界面，用于向用户展示数据以及接收用户的输入

而Model层就是JavaBean实体类，用于保存实例数据

Controller控制器用于更新UI界面和数据实例

# MVP



MVP中Activity只处理界面，把业务逻辑抽象成Presenter接口，Model类还是原来的Model。

View并不能直接对Model进行操作，这也是MVP与MVC最大的不同之处。

优点：

1、分离了视图逻辑和业务逻辑，降低了耦合

2、Activity只处理生命周期的任务，代码变得更加简洁

3、视图逻辑和业务逻辑分别抽象到了View和Presenter的接口中去，提高代码的可阅读性

4、Presenter被抽象成接口，可以有多种具体的实现，所以方便进行单元测试

5、把业务逻辑抽到Presenter中去，避免后台线程引用着Activity导致Activity的资源无法被系统回收从而引起内存泄露和OOM

# MVVM

Databinding实现vm层，activity只关注UI逻辑，数据直接从vm绑定到xml文件，方便保存页面数据

View修改数据会同步到Model，当Model变化时，View-Model会自动更新，View也会自动变化。

# 排序算法

## 快速排序

通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以[递归](https://baike.baidu.com/item/%E9%80%92%E5%BD%92/1740695)进行，以此达到整个数据变成有序[序列](https://baike.baidu.com/item/%E5%BA%8F%E5%88%97/1302588)。

## 冒泡排序

比较相邻两个数大小，前一个数大则交换，后一个数大则用后一个数再与下一个数比较，比较到最后一个数为一轮。

for(int i=1;i<arr.length;i++){

for(int j=1;j<arr.length-i;j++){

//交换位置

}

## 选择排序

每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，直到全部待排序的数据元素排完。选择排序是不稳定的排序方法。

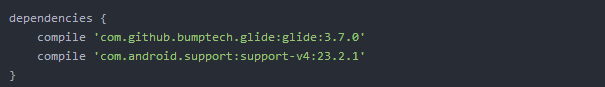
遍历比较相邻两数大小，记录下最小数位置，把最小数交换到第一位。

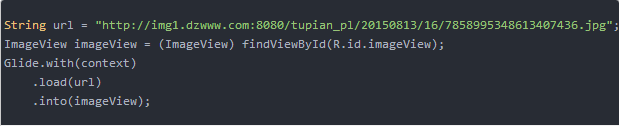
## 插入排序

# 图片加载框架

## Glide

glide可以支持gif和短视频









## Picasso





## Fresco、BitmapUtils、ImageLoad

# React Native

React Native使你只使用JavaScript也能编写原生移动应用。

React Native产出的并不是“网页应用”，或者说“HTML5应用”，又或者“混合应用”，最终产品是一个真正的移动应用。

React Native让你可以快速迭代开发应用。 比起传统原生应用漫长的编译过程，现在你可以在瞬间刷新你的应用。编译很快。

能生成IOS和android。

# JVM

Dalvik虚拟机允许多个instance的存在。实际上android中的每一个app都是运行在自己VM实例之中(沙盒)。每一个VM实例在linux中又是一个单独的进程，所以可以认为是同一个概念。运行在自己的DVM进程之中，不同的app不会相互干扰，且不会因为一个DVM的崩溃导致所有的app进程都崩溃。

# Java四种引用

1、强引用：创建一个对象并把这个对象直接赋给一个变量，eg ：Person person = new Person(“sunny”); 不管系统资源是否紧张，强引用的对象都绝对不会被回收，即使他以后不会再用到。只有在赋值为null的时候会回收。

2、软引用：通过SoftReference类实现，eg : SoftReference p = new SoftReference(new Person(“Rain”));内存非常紧张的时候会被回收，其他时候不会被回收，所以在使用之前要判断是否为null从而判断他是否已经被回收了。

3、弱引用：通过WeakReference类实现，eg : WeakReference p = new WeakReference(new Person(“Rain”));不管内存是否足够，系统垃圾回收时必定会回收

4、虚引用：不能单独使用，主要是用于追踪对象被垃圾回收的状态，为一个对象设置虚引用关联的唯一目的是希望能在这个对象被收集器回收时收到一个系统通知。通过PhantomReference类和引用队列ReferenceQueue类联合使用实现

# 事件分发机制

ViewGroup的相关事件有三个：

dispatchTouchEvent、onInterceptTouchEvent、onTouchEvent。

View的相关事件只有两个：

dispatchTouchEvent、onTouchEvent。

dispatchTouchEvent该方法用来分发事件，一般不会重写这个方法，（ViewGroup可以用onInterceptTouchEvent返回true拦截，view只能在onTouchEvent处理返回true，无法再放子view传递）   
onInterceptTouchEvent用来拦截事件，返回true不再向子控件传递onTouchEvent用来处理事件，返回true表示处理完成

Touch事件分发会一层一层向子控件传递，onInterceptTouchEvent返回true可拦截事件不再向下传递，直接由该ViewGroup处理（onTouchEvent）。onTouchEvent返回false表示没有处理完这个事件，事件将由子控件向上层控件一层一层处理，若返回true，则不再传递到上层控件处理。

# 性能优化

## 启动速度优化

1第三方库初始化不必要放在application的onCreate中，可以在使用到的时候再初始化

2首页布局层级复杂度影响，减少测量绘制时间

3别做耗时操作

4为启动activity设置style的windowBackground背景图片（闪屏）防止启动黑白屏

## 安装包大小优化

1as可以查找无用资源

2图片格式使用jpg和webp代替png，减小图片大小

3删除无用的代码，并混淆

4去掉无用依赖库

## 卡顿优化

1优化UI布局层级

2主线程不做耗时操作

## 内存优化

1使用LeakCanary，运行app时发生内存泄漏时会通知提示哪个Acvitity/Fragment有内存泄漏。修改过再运行该类不再报泄漏则已经正常。使用起来比较直观。

2 使用AS自带的Android Profile（android monitor）监控内存增长，给什么对象分配了多少内存，也可以查看cpu和网络使用情况。点击record和stop监控一段时间内的变化。

# 加密

Base64编码一般用于url的处理，编码方式，编码算法完全公开，可解密

MD5定长，可解密

SHA定长，只能加密

DES对称加密，加密和解密都使用同一个密钥

AES对称加密，加密和解密都使用同一个密钥

RSA非对称加密，需要两个密钥来进行加密和解密，分别是公钥和私钥。

# Kotlin优势

代码简洁，提高效率，可以和java代码互相调用

构造函数可默认值，很多时候不必写多个重复参数构造函数

添加了拓展函数，如String.substringBefore()，提高效率。可自定义拓展函数，为特定类扩展新的方法。

接口可提供抽象和非抽象方法，抽象属性需要在实现类的构造函数中赋值

属性定义时就定义了可变或不可变，空或非空，利于调用的时候检查是否为空，减免了空指针可能性

Kotlin集合分为只读和可变，使程序容易理解

拓展函数生成一个java的静态方法，将调用的对象作为一个参数传入。

伴生对象companion object，默认名Companion，可以起名如companion object xx。它是一个普通对象，相当于静态方法，可调用私有构造函数

匿名对象，object：实现接口

可以使用lambda风格编写代码

as? 安全判断

?. 不为空时获取值

.let 方法不可传空而传入对象可空时可用，需要传入多个参数还是用if来判断比较简洁

?: 不为空执行冒号前面，为空执行后面

with(a){} 可直接调用对象a的方法执行操作

apply{} 对象转化为this进行操作，最后会return a

run{} 对象转化为this，返回最后一行

let{} 对象转化为it，返回最后一行

filter() 过滤并留下表达式中为ture的元素

map() 会将选出符合条件元素中的属性变成一个新集合，比如将学生集合变为学生年龄的集合

all() 判断集合中所有元素是否都满足条件

any() 判断集合中有元素满足条件

count() 统计集合满足条件元素个数

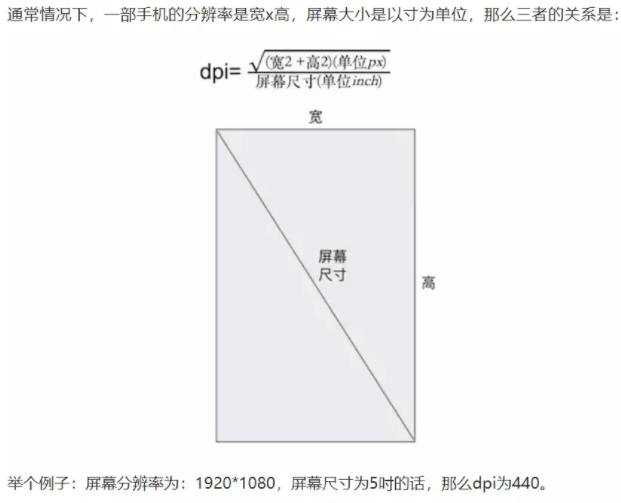
find() 找出集合第一个满足条件的元素

asSequence() 序列，filter和map会返回一个新列表，调用后不再对filter和map直接生成列表，提高效率

Any类似Java的Object，是所有类型的根，并包括了基础数据

.filterNotNull() 过滤掉集合中为null的元素

# 屏幕适配



smallestWidth 按照一个最小dp为基准为1生成多个value-swxxxdp（xxx数值跟尺寸、dpi相关）文件对应不同dimens.xml文件，其他dp值将按最小基准dp调整相应倍数，布局中引用相应的值适配不同dp（px/（dpi/160））

dpi和尺寸成反比，和总dp成反比。尺寸越大总dp越大，布局文件的dp越大。dpi越小总dp越小，布局文件的dp越小。

举个例子，小米5的dpi是480,横向像素是1080px，根据px=dp(dpi/160)，横向的dp值是1080/(480/160),也就是360dp,系统就会去寻找是否存在value-sw360dp的文件夹以及对应的资源文件。

缺点耦合性大，把dp换成参数值，更换框架需要修改大量布局文件。资源文件多占空间。

今日头条 不同手机厂家设置的dpi不同，最终宽高换算成dp不同（将dp计算方式px/（dpi/160）=dp变为px/density=dp）。设定设计图尺寸dp，将px和dpi固定值dp变化值改为px和dp固定值density为变化值，使得density的值相应变化无视厂家设定的dpi值，布局时通过计算动态以设计图dp为基准调整实际宽高px和总px的对应比例值相同。

相同分辨率因为dp相同，转换成的像素值相同，固定了density 不考虑dpi不会受到尺寸影响。

设备一

屏幕总宽度为 1080 px，根据今日头条的的公式求出 density，1080 / 375 = 2.88 (density)

这个 50dp \* 50dp 的 View，系统最后会将高宽都换算成 px，50dp \* 2.88 = 144 px (根据公式 dp \* density = px)

144 / 1080 = 0.133，View 实际宽度与 屏幕总宽度 的比例和 View 在设计图中的比例一致 (50 / 375 = 0.133)，所以完成了等比例缩放

设备二

屏幕总宽度为 1440 px，根据今日头条的的公式求出 density，1440 / 375 = 3.84 (density)

这个 50dp \* 50dp 的 View，系统最后会将高宽都换算成 px，50dp \* 3.84 = 192 px (根据公式 dp \* density = px)

192 / 1440 = 0.133，View 实际宽度与 屏幕总宽度 的比例和 View 在设计图中的比例一致 (50 / 375 = 0.133)，所以也完成了等比例缩放

缺点在统一设计图尺寸上调整，第三方控件设计图尺寸会与设定不一致。

# 屏幕尺寸

1. mdpi[320x480px]

2. hdpi [480x800px、480x854px、540x960px]

3. xhdpi[720x1280px]，4.5-6寸

4. xxhdpi[1080x1920px]，[1440x2560px]

L DPI ( Low Density Screen，120 DPI )，其图标大小为 36 x 36 px

M DPI ( Medium Density Screen, 160 DPI )，其图标大小为 48 x 48 px

H DPI ( High Density Screen, 240 DPI )，其图标大小为 72 x 72 px

XH DPI ( Extra-high density screen, 320 DPI )，其图标大小为 96 x 96 px

XXH DPI( xx-high density screen, 480 DPI )，其图标大小为144 x 144 px

XXXH DPI( xxx-high density screen, 640 DPI )，其图标大小为192 x 192 px

# app保活

## 广播

注册静态广播监听，利用一个app广播启动另一个app。利用系统广播打开app。

## 通知

前台service，开启通知告知用户可点击打开app。

## 前台Service（灰色保活）

Service优先级很低，会被系统杀死。前台service，不会由于系统内存不足的原因导致被回收，会一直有一个正在运行的图标在系统的状态栏显示，下拉状态栏后可以看到更加详细的信息，非常类似于Notification的效果。启动前台service可隐藏掉Notification达到。

# Room数据库

[Room](https://developer.android.google.cn/training/data-storage/room/index.html)持久库提供了一个SQLite抽象层，让你访问数据库更加稳健，提升数据库性能。

使用注解编写查询语句和字段。

可以配合Rxjava使用

# Include、Merga和ViewStub的区别

https://blog.csdn.net/guolin\_blog/article/details/43376527

<include>标签可以允许在一个布局当中引入另外一个布局，一般封装一个title来include使用。

<merge>标签是作为<include>标签的一种辅助扩展来使用的，它的主要作用是为了防止在引用布局文件时产生多余的布局嵌套。替换掉原布局中的xxxlayout标签，减少布局嵌套。Include到约束布局中可先进行约束，但无法预览到布局效果，需要在Include预览布局效果。

ViewStub是动态布局标签，引入的布局内的控件需要ViewStub.inflate()才会显示，否则不会绘制viewStub里的控件。

# 待更新知识点

泛型、重写、重载、锁synchronized、jni、全局变量线程池、数据结构、不赋予读写权限怎么写文件、序列化、RXjava、数据结构、图片管理、内存泄漏检测

# 面试题链接

https://www.jianshu.com/p/7bb56481063d