1. ANR定位：Bugly、/data/anr/tarce文件

ANR时间、进程名、进程PID、线程名、EventLog、堆栈信息

提问：线上OOM 你们hropf文件怎么上传的?

2. WebView进程抽离：

1. IPC-Bindler、ContentProvider

2. callback回调对象注册后无法解绑 - 序列化和反序列化；

3. 多进程重复初始化Application - 初始化前判断进程名；

4. 本地线程和bindler线程问题 - 处理好线程交互；

5. WebView每次打开进程过慢 - 开启App或首次打开WebView时创建一个独立经常并开启一个Service并维护他

6. 持久化数据（Cookie）；进程间通过ContentProvider持久化数据；

7. 用户登陆状态

Binder核心(效率、稳定性)：接收方缓存区与内核缓存区是映射到同一块物理地址，只需要发送进程将数据拷贝至内核缓存区，减少了一次数据拷贝过程。

效率仅次于共享文件(但共享文件致命点是无法有效处理同步问题/资源死锁问题)比Socket、管道、消息队列（两次数据拷贝）更高效。

ServiceManager：Server端向Binder驱动注册服务，Binder驱动会将服务同步到ServiceManager上，这时ServiceManager上拥有Server进程信息；

之后Client向Binder驱动申请服务时会转发到ServiceManager上，找到对应Server服务并返回引用给Client，Client；

注册服务、获取服务、使用服务(内存映射-一旦发送进程发送数据到内核缓存区时，立即同步到接收进程)；

AIDL：Android接口定义语言，Android提供的一种进程间通信机制。本质上是基于Binder，通过在Android编译时自动生成包含Stub和Proxy的静态内部类的AIDL文件，简化我们binder进程间通信的实现；

3. ReentrantLock - 可重入锁；

CountDownLatch - 计数器锁；countDown-1；0时触发.await()等待继续执行；

BlockingQueue - LinkedBlockingQueue（一个由链表结构组成的有界队列，此队列的长度为Integer.MAX\_VALUE。此队列按照先进先出的顺序进行排序；FixedThreadPool、SingleThreadExecutor）

、PriorityBlockingQueue（一个支持线程优先级排序的无界队列，默认自然序进行排序，也可以自定义实现compareTo()方法来指定元素排序规则，不能保证同优先级元素的顺序）

、DelayQueue（一个实现PriorityBlockingQueue延迟获取的无界队列，在创建元素时 可以指定多久才能从队列中获取当前元素。只有延时满后才能从队列中获取元素；ScheduledThreadPool）

、SynchronousQueue（一个不存储元素的阻塞队列，每一个put操作必须等待take操作，否则不能添加元素；CachedThreadPool）

4. 死锁四条件：互斥、请求与保持、不剥夺、循环等待；

5. ANR条件：UI线程(布局加载、耗时操作)耗时、子线程同步锁导致UI线程等待、binder线程满导致UI线程无法与SystemServer通信、对端等待(IPC-binder对端、ContentProvider、Socket)；

ANR基本分析流程：

1.抓取Trace堆栈信息，搜索ANR in，查看发生的时间和进程

2.根据进程寻找主线程的trace，发现被blocked的地方

3.根据具体被Blocked的类和行数，结合源码进行分析解决

6. App安全：

App代码安全：混淆、加密、加壳；

App数据存储安全：本地磁盘做数据持久化时所做的加密处理；

App网络传输安全：Client 到 Server 的数据报文加密，防止网络传递过程中被抓包窃取数据；

加密算法：

对称加密算法：AES --加密、解密使用同一密钥，效率高，但安全性低；

非对称加密算法：RSA（HTTPS-“加密”“签名”）、ECC --加密、解密使用不同密钥，效率低、但安全性高； 私钥（持有人、不暴露）、公钥（所有请求者）

电子签名，确认通信双方

普通加密算法：MD5、SHA

7. 启动流程关键：

Activity.startActivity() -> Instrumentation.execStartActivity() -> ActivityManagerNative.getDefault().startActivity() -> ActivityManagerProxy.startActivity()

远程(SystemS)：ActivityManagerService、ActivityManagerNative 本地：ActivityManagerProxy 在

远程(本地)：ApplicationThread、ApplicationThreadNative 本地(SystemS)：ApplicationThreadProxy 在main方法的thread.attach(false)上关联远程与本地；

handleLaunchActivity() -> performLaunchActivity() --a.attach() {PhoneWindow、WindowManager -- mWindow.setWindowManager(xx)关联}}

handleResumeActivity() --wm.addView(decor,l) -> viewRoot.setView() -> doTraverse()绘制View树

扩展：ServiceManager（查询、注册 远程Binder服务），init进程初始化ServiceManager时会初始化一系列系统服务，Zygote进程创建SystemServer时也会将各个系统服务注册到ServiceManager上；

sm.addService() / sm.getService();

8. JNI与NDK

Java调C/C++:Java层定义具体Native方法，然后在底层实现对应.h文件(定义对应方法)，然后实现编写c/cpp文件实现这些具体方法，通过gcc指令打包成.so库，传到应用层上就能调用具体Natice方法。

C/C++调Java方法：现在Java本地定义好具体方法，在C/C++上通过findClass找到具体类名，然后通过GetStaticMethodId找到具体方法ID，配置参数，通过CallStaticVoidMethod(clazz,id,agrs..)调用;

9. okHttp和Volley对比：

Volley：支持Https、缓存、异步请求、本身不支持同步请求，支持协议有Http1.0/1.1；网路传输底层采用HttpUrlConnection数据读写采用IO；

okHttp：支持Https、缓存、异步请求、同步请求、支持协议有Http1.0/1.1/2.0、SPDY，网路传输底层采用的是封装的Socket，数据读写是NIO；

SPDY协议类似Http，目的是为了缩短加载时间和提高安全性，SPDY通过压缩、多路复用和优不同先级来缩短加载时间；

10.Glide核心： --Glide.with(this).load(imageUrl).override(800, 800).placeholder().error().animate().into()

.with: 初始化各式各样的配置信息（包括缓存，请求线程池，大小，图片格式等等）以及glide对象;

.load: 配置请求URL，并记录URL已记录状态。

.into: 根据转码类transcodeClass类型返回不同的ImageViewTarget：BitmapImageViewTarget、DrawableImageViewTarget；设置缩略图、根据设定的宽高执行engin.load()加载图片

Engine是一个负责加载和管理图片缓存资源的类：

常规三级缓存：强引用(LruCache容器)、软引用(SoftReference容器)、磁盘缓存(DiskLruCache容器)；

Glide三级缓存：内存缓存(MemoryCache/LruCache)、弱引用容器(activeResources--Map<Key, WeakReference<EngineResource<?>>>)、磁盘缓存；

取过程：内存MemoryCache、弱引用容器、磁盘

LruCache若存在，则返回并移除图片对象,并将该对象存入弱引用容器activeResources(该容器存放正在使用的图片对象),若没用时在移除该对象并存放入LruCache中；最后异步请求(网络/磁盘)

存过程：弱引用容器、内存、磁盘

同理，在网络上下载完图片对象之后，首先存放到弱引用容器activeResources，然后存放至磁盘diskLruCache中，之后不用时存入LruCache。

内存缓存基于LruCache实现，磁盘缓存基于DiskLruCache实现。这两个类都基于Lru算法和LinkedHashMap来实现。

LRU是Least Recently Used的缩写，最近最少使用算法，它的核心原则是如果一个数据在最近一段时间没有使用到，那么它在将来被访问到的可能性也很小，则这类数据项会被优先淘汰掉。

11.bitmap压缩策略： 图片占用内存的计算公式：图片高度 \* 图片宽度 \* 一个像素占用的内存大小

BitmapFactory 四类方法：

decodeFile( 文件系统 )

decodeResourece( 资源 )

decodeStream( 输入流 )

decodeByteArray( 字节数 )

关键参数：BitmapFactory.options

inSampleSize 采样率，对图片高和宽进行缩放，以最小比进行缩放（一般取值为 2 的指数）

inJustDecodeBounds 仅获取图片的宽高信息，交给 inSampleSize 参数选择缩放比

A：Options.inPurgeable：设置为True时，表示系统内存不足时可以被回收，设置为False时，表示不能被回收。

B：Options.inInputShareable：设置是否深拷贝，与inPurgeable结合使用，inPurgeable为false时，该参数无意义。

高效加载 Bitmap 的流程:

1、将 BitmapFactory.Options 的 inJustDecodeBounds 参数设为 true 并加载图片；

2、从 BitmapFactory.Options 中取出图片原始的宽高信息，对应于 outWidth 和 outHeight 参数；

3、根据采样率规则并结合目标 view 的大小计算出采样率/压缩比 inSampleSize；

4、将 BitmapFactory.Options 的 inJustDecodeBounds 设置为 false 重新加载图片(此时加载的图片像素与ImageView大小匹配)；

12.内存泄漏/内存溢出分析：

内存泄漏：通过Memory Profiler或Memory Monitor工具对内存进行堆转储操作，生成对应的HProf文件(类实例信息、对象的引用树)，然后通过我们的分析工具进行分析；

内存溢出：一般通过Bugly线上分析工具对OOM异常进行定位和日志跟踪，高版本系统需要拿到用户Trace文件进行堆栈信息日志跟踪(进程号、时间、堆转信息)；

内存的真正释放和对象的生命周期有关也和GC的调度有关；

LeakCanary：内存泄漏

思想：Activity执行完onDestroy()之后，将它放入WeakReference中，然后将这Activity对象与ReferenceQueque(存放已被GC的对象)关联，GC后再查看RefQue是否有销毁的对象，若无，这内存泄漏；

BlockCanary:卡顿监听

思想：通过Handler消息机制原理，再Msg被Looper轮询到之后，根据Msg.target 将消息dispatchMessage分发到对应Handler，在dispatch阶段记录时间，统计我们Msg的执行时间，超过阈值则卡顿；

13.Bugly及Trace日志监听

Bugly：异常、奔溃、卡顿、ANR、错误...

Trace：ANR异常， 策略-首次启动上传、定时上传、ANR监听上传...

14.RESTful API

流行的API接口设计规范，给后台API接口定义了一种设计原则和约束条件；

15.ADB与Shell脚本：

ADB：调试桥工具，即Debug工具，PC与Android手机之间交互桥梁。通过ADB我们可以管理我们设备或模拟器的状态，如APK安装、卸载、进入设备运行Shell命令等等；

常用： adb devices(查看当前连接设备)

adb install <apk路径>(安装APK)

adb unstall <包名>(卸载APK)

adb reboot(重启)

adb shell(进入设备Shell环境)

adb shell ps/top (查看进程信息)

adb shell am start <apk包名>/<activityName> (am即ActivityManager，通过am模拟系统行为)

adb shell pm list packages (pm即PackageManager，通过pm模拟系统行为)

Shell：上层用户与设备系统交互的桥梁，提供一个Shell环境来操控设备系统。

常用： chmod (改变文件权限)

cd /data/local (进入指定目录)

cd / (进入根目录)

cat (查看文件信息)

cp (拷贝文件)

mv (移动文件)

ls (列出目录内容)

mkdir (创建文件夹)

touch (创建文件)

rm (删除文件)

rmdir (删除文件夹)

echo xxx >>1.txt (输出内容至1.txt)

ps -ef (查看进程信息)

ps -ef/grep xx (查看指定进程)

kill <pid> (杀死进程)

16、SQLite数据库：

17、平衡二叉树(二路)、B-Tree(多路)、B+Tree

平衡二叉树特点：非叶子节点最多允许2个子节点、树的左右层级差距不大于1、左小右大原则、当层级失衡时会自动调整(LL\RR\RL调整算法)

B-Tree：和平衡二叉树类似，只是非叶子节点的子节点能大于2；树的所有叶子节点都在同一层级、叶子节点包含关键字及指针(只是叶子指针指向为Null)、关键字树大于路数则调整(拆分排序)

B+Tree：升级版的B-Tree，效率更高更稳定。同样平衡树、非叶子节点不保留

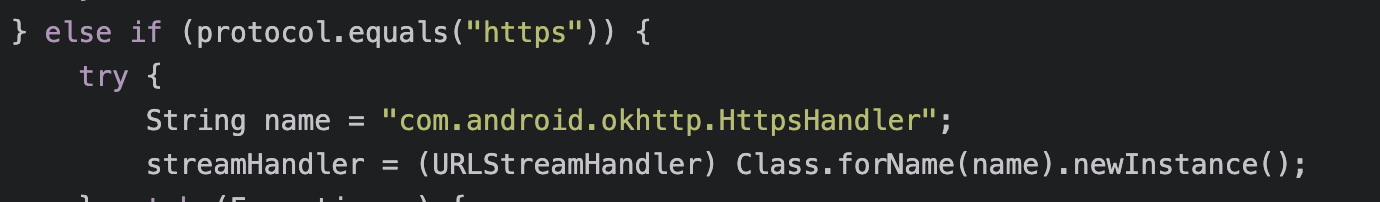
数据库索引技术大量采用B-Tree、B+Tree；

思路策略都是相同：通过二分法和数据平衡策略避免严重的树形偏离，保证节点查找速度。

每次策略调整都是为了让节点利用更加合理，最大限度减少层级结构。

1. Android 4.4版本以上的HttpURLConnection底层是用OkHttp实现的：

streamHandler.openConnection(this);



1. **Android程序员入坑Flutter的大体知识要点：**

**<https://www.jianshu.com/p/3eff00f07008>**

**<https://blog.csdn.net/weixin_44339238/article/details/108987590>**

1. **Fluuter和Jetpack Compose的优劣势：**

**Fluuter优势：**

1. **基于渲染引擎Skia，支持同时部署到两端设备上且无差异，节省开发时间。**
2. **基于组件嵌套来实现布局、免除Layout文件的写法。**
3. **支持Hot Reload热重载，避免编译等待，提高开发效率。**
4. **丰富的UI组件库，能够快速构建应用。**

**Fluuter劣势：**

1. **糟糕的UI控件API，比如想在代码上动态增删部分UI，都得重新走build流程，而且自定义View也很不方便，基本只能在原组件库中找实现。**
2. **因为Fluuter的页面布局是基于组件开发的，“万物皆组件”遇到一些稍微复杂些的页面，组件嵌套层级会非常多，这种层级嵌套可能在前期开发的时候效率会比较快，当时到中后期的迭代跟维护上就很大坑了，阅读成本和维护成本也大大提高了，**
3. **而且Flutter不像HTML，把组件的样式都交给CSS操作，而是样式和定义都混在一起，导致代码非常庞大。**

**总结：Flutter的优势在于多端支持，劣势在于组件层级太多，后期迭代维护成本高。**

1. **IntentService和HandlerThread区别。**

**1、IntentService本质上是一个Service，只是在Service基础上做了线程的封装（底层通过HandlerThread+Handler封装），实现onHandleIntent() 在后台执行异步任务，执行完成后自动销毁服务。**

**2、HandlerThread本质上是一个线程类，只是在Thread内部做了Looper的初始化，并提供子线程的Handler封装。**

**3、Handler执行HandleMessage()的线程是创建Looper的线程。**

1. **Service和Broadcast都是执行在Main线程，耗时操作用IntentService，或继承IntentService类写相关方法（持续不关闭StopSelf()）。**
2. **IntellJ+JFrame+ApectJ，注解-预编译生成-统一维护-无运行时性能影响、业务与切面埋点逻辑隔离。 注解作用标记识别相关方法。**



1. **Tencent-GT 性能检测：**

<https://blog.csdn.net/TMQ1225/article/details/79498468>

<https://blog.csdn.net/tmq1225/article/details/79529230>

<https://blog.csdn.net/TMQ1225/article/details/79568855>