# 数字座舱系统 Android 端性能优化

文档修订记录

| 序号 | 日期 | 修改人 | 修改内容描述 | 版本号 |
|---|---|---|---|---|
| 1 | 2022/07/05 | 刘昆明 | 初稿 | 1.0 |

# 1 前言

## 1.1 Android 端性能优化简介

数字座舱系统Android端的性能优化主要包括启动耗时优化，内存优化，卡顿优化，稳定性优化等。本文对Bigsur，Monza 等项目的启动耗时，内存优化等性能优化进行总结，重点是 **Android Java** 层的性能优化，形成平台化文档，用于其他项目中做 Android 端性能优化参考。

## 1.2 系统性能优化方法和工具

| 优化类型 | 工具 |
| --- | --- |
| 启动耗时优化 | Google 优化启动文档 中提到使用 bootchart，Systrace 等工具优化启动时间；<br>用于转储系统消息日志的命令行工具：Logcat；<br>查看系统启动：cat /sys/kernel/boot_kpi/kpi_value； |
| 内存优化 | dumpsys meminfo, top, memory_profiler, Memory Analyzer (MAT), LeakCanary |
| 流畅度/卡顿优化 | systrace，cpu profiler |
| ANR 稳定性优化 | trace，Logcat |
| Crash 稳定性优化 | Logcat (Java)，stack --syms (Native) |

# 2 启动耗时优化

## 2.1 Persistent 白名单配置

Android 中的 Persistent App（介绍，原理）具有两个特性：

- 在系统刚起来的时候，该 App 也会被启动起来；
- 该 App 被强制杀掉后，系统会重启该 App。这种情况只针对系统内置的 App，第三方安装的 App 不会被重启。

Android 系统启动时，在 system_server 启动初始化中，通过调用 ActivityManagerService (AMS) 的 systemReady 方法，在systemReady 中会调用 startPersistentApps 自动启动这些 Persistent App。

但系统中的部分 Persistent Apps 和第三方 Apps 可以在 system_server 启动后延迟启动，因此，对部分需要开机后在 startPersistentApps 中立即启动的 apps 增加白名单配置。Framework AMS 的改动：
https://git.i-tetris.com/c/qcom/sa/quic/la/platform/frameworks/base/+/88736

白名单配置(以 Bigsur 为例，各项目定制具体白名单 )：
`device/mega/bigsur/config/non-delay-persistent-apps.xml`
说明：enable 指是否打开 persistent 白名单 feature 开关，interval 指延迟启动的时间，单位：ms

## 2.2 Launcher 延迟广播

在 Android 启动时，对于需要开机自动启动、但可以在 Launcher 启动后再启动的 Apps，可以在 Launcher 中增加延迟广播。这些 Apps 通过监听这个广播来启动自己，这样可以减少 Android 系统启动时这些 App 对系统内存、CPU 等资源的占用，优化优先级更高的 Apps(如：Launcher，导航等)的启动耗时。

- Launcher 中实现延迟广播代码参考：
  https://git.i-tetris.com/c/cores/gladius/android/apps/MegaLauncher/+/84427
- App监听此广播来启动代码参考：
  https://git.i-tetris.com/c/apps/gladius/android/changan/c385/BigsurRepair/+/82952

## 2.3 模块去除

对于有镁佳定制实现或第三方实现的 Apps(如：CarSettings，输入法等)，可以去掉 Google 原生的模块，AOSP 中的 test 目录中的部分模块，和 megaos 中不需要的其他模块，都可以通过配置 packages_remove.mk 去除。packages_remove.mk路径：
`device/mega/<项目名>/packages_remove.mk`
其中，<项目名> = monza, bigsur, …

## 2.4 去掉 keyguard, wallpaper 等开机时的检查

代码改动参考：
https://git.i-tetris.com/c/qcom/sa/quic/la/vendor/mega/device/bigsur/+/75224

# 3 内存优化

## 3.1 Android 系统内存优化

### 3.1.1 多用户配置

Android 10 之后默认启动了多用户。启动多用户后，如果 App 不针对多用户进行相应的修改，则可能会在每个用户下都有一份应用的进程，造成内存占用的浪费。

1）配置系统应用安装到指定用户

- 在 device/mega/<项目名>/ 下创建 preinstalled-packages-config 目录
- 在 device/mega/<项目名>/preinstalled-packages-config 下放入配置文件preinstalled-packages-<项目名>.xml。如：<项目名> = monza, bigsur, …，在preinstalled-packages-monza.xml 配置需要运行在指定用户下的 App。关于多用户下的 userType 定义参考 Google官方文档，定义在代码 frameworks/base/core/java/android/os/UserManager.java 中的。其中，user-type="FULL" 指 App 运行在 U10，U11…，user-type="SYSTEM" 指 App运行在 U0。预安装相关的知识可以参考 Google预安装相关官方文档。

2）调用多用户 API 启动特定用户的进程

```
Context.startActivityAsUser(Intent, UserHandle)
Context.bindServiceAsUser(Intent, ..., UserHandle)
Context.sendBroadcastAsUser(Intent, ... , UserHandle)
Context.startServiceAsUser(Intent, ..., UserHandle)
```

3）调用 ContentObserver 和 BroadcastReceiver 上的新 API 监听特定或所有用户以及回调

```
MegaContentResolver.registerContentObserver(Uri uri, boolean
        notifyForDescendents, ContentObserver observer,
        @UserIdInt int userHandle)

MegaContext.registerReceiverAsUser(Context context,
        BroadcastReceiver receiver,
        UserHandle user, IntentFilter filter,String broadcastPermission,
        Handler scheduler)
```

4）使用单例

将 android:singleUser="true" 加入到服务、接收器或程序的 AndroidManifest.xml 中，则系统只在 U0 下实例化一个服务，接收器或者程序。

### 3.1.2 去掉不需要的模块

参见 第2.3章

## 3.2 App 内存优化

背景阅读：Android 官方内存相关的技术文档：概览，内存分配，管理应用内存。

内存优化心得分享还可以参考：数字座舱系统 Android 内存优化指南。

### 3.2.1 数据结构优化

1）SparseArray 系列代替 HashMap
当数据量小于1000，Key 为 int 类型或 long 类型，可以考虑用 SparseArray 系列代替 HashMap。SparseArray 系列有 SparseArray，SparseBooleanArray，SparseIntArray，SparseLongArray，LongSparseArray。

2）ArrayMap 代替 HashMap
HashMap 的 Key 为其他类型，并且数据量小于 1000 时，可以考虑用 ArrayMap 代替 HashMap

3）Message.obtain() 代替 new Message()

4）Parcelable 代替 Serializable
Serializable 的序列化和反序列化都需要使用到 IO 操作，而 Parcelable 不需要 IO 操作，Parcelable 的效率要高于 Serializable，Serializable 在序列化的时候创容易触发 gc。Android 中内存间数据传输时推荐使用 Parcelable。

### 3.2.2 类优化

1）RelativeLayout 或者 ConstraintLayout 代替用 LinearLayout
LinearLayout 中布局嵌套较深时，容易引起 ANR。可以使用 RelativeLayout 或者 ConstraintLayout 代替用 LinearLayout 减少嵌套层数，优化 App 流畅度。

2）IntentService 代替 Service
IntentService 使用队列的方式将请求的 Intent 加入队列，然后开启了一个 Worker Thread（工作线程）在处理队列中的 Intent。对于异步的startService请求，IntentService 会处理完成一个之后、再处理第二个，每一个请求都会在一个单独的 Worker Thread 中处理，不会阻塞应用程序的主线程。如果我们如果要在 Service 里面处理一个耗时的操作，我们可以用 IntentService 来代替使用。

3）RecyclerView 代替 ListView
在需要动画，或频繁更新，局部刷新等场景，建议使用 RecyclerView 代替 ListView。

### 3.2.3 内存泄漏优化

#### 3.2.2.1 内存泄漏排查

1. `dumpsys meminfo <package_name>` 查看 PSS 等内存是否一直增长

2. MEM 趋势线
3. memory profiler 查看进程的内存变化
4. 集成 LeakCanary 监控 App 的 Activity/Fragment 是否有内存泄漏

#### 3.2.2.2 内存泄漏原因和类型

内存泄漏原因：应该被释放的对象直接或间接被 GC root 引用，或者生命周期较长的对象持有生命周期较短的对象的引用。

GC root 类型：

1. 虚拟机栈（栈帧中的本地变量表）中引用的对象
2. 本地方法栈中 JNI（即一般说的 Native 方法）引用的对象
3. 方法区中类静态属性引用的对象
4. 方法区中常量引用的对象

内存泄漏常见类型和解决方案：

1）集合类

- 内存泄露原因：集合类添加元素后，仍引用着集合元素对象，导致该集合的元素对象不可被回收，从而导致内存泄漏；
- 解决方案：集合类添加集合元素对象后，在使用后必须从集合中删除，所有元素不需要使用时清空集合对象 & 设置为 null。

2）持有 Context 造成的内存泄漏

- 内存泄露原因：当我们给一个类传递 Context 时候，如果 Context 是 Activity 类型，这就导致了该类持有对 Activity 的全部引用，当 Activity 关闭的时候，因为被其他类所持有，而导致无法被正常回收，而导致内存泄漏；
- 解决方案：在给其他类比如单例传递 Context 的时候使用 Application 对象，这个对象的生命周期和整个应用相同，而不依赖 Activity 的生命周期，而对 Context 的引用不要超过他本身的生命周期，谨慎对 Context 使用 static 关键字。

3）Handler 造成的内存泄漏

- 内存泄露原因：如果 Handler 是非静态内部类或非静态匿名内部类，会默认持有外部类（比如：Activity 的引用）。当这个 Activity 退出时、消息队列中还有未处理或者正在处理的消息，而消息队列中的 Message 持有对 Handler 实例的引用，Handler 又持有 Activity 的引用，所以导致该 Activity 的内存资源无法及时回收，引发内存泄漏。

- 解决方案：
    a. 把 Handler 放到单独的类中，或者使用静态的内部类（静态的内部类不会引用 Activity）避免泄漏；
    b. 如果想在 Handler 内部去调用 Activity 中的资源，可以在 Handler 中使用弱引用的方式指向所在 Activity，即用 static+weak reference断开 Handler 与 Activity 的强引用关系。

4）非静态内部类创建静态实例造成的内存泄漏

- 内存泄漏原因：非静态的内部类默认会持有外部类的引用，又使用非静态内部类创建了一个静态实例，该静态实例的生命周期和应用一样长，这就导致了该静态实例一直会持有该 Activity 的引用，导致 Activity 不能正常回收；
- 解决方案：将非静态内部类改成静态的，这样他对外部类就没有引用或者将该对象抽取出来封装成一个单例。

5）线程造成的内存泄漏

- 内存泄漏原因：异步任务 AsyncTask 和 Runnable 都是匿名内部类，因此他们对当前 Activity 都有一个隐式的引用。如果 Activity 在销毁之前，任务还没有完成，那么将导致 Activity 的内存资源无法回收，造成内存泄漏；
- 解决方案：使用静态内部类，在 Activity 销毁时候也应该取消相应的任务 AsyncTask::cancel()，避免任务在后台执行浪费资源。

6）资源未关闭造成的内存泄漏

- 内存泄漏原因：对使用了 BroadcastReceiver，contentObserver，File，Cursor，Stream，Bitmap等资源的代码，在 Activity 销毁时未及时关闭、unRegister或者注销；这些资源将不会被回收，造成内存泄漏。
- 解决方案：对使用了 BroadcastReceiver，contentObserver，File，Cursor，Stream，Bitmap等资源的代码，应该在 Activity 销毁时及时关闭、unRegister或者注销。
- 举例：BGS-21155 / BGS-22579。
  问题分析：开发在使用 *ImageView* 或 *View*背景设置的时候，没有频繁变化的场景需求，因此使用的都是基本方法，类似 *setImageResource*，*setBackground* 使用*drawable*。例如：
  *setImageResource* 是从资源 *drawable* 中通过资源 *id* 找到文件转成可绘制对象 *drawable* 然后绘制。这个方法会自动适配分辨率。适用于不频繁设置图片图片资源不会太大的情况。但是对于大图片时或者需要不断的重复的设置图片，调用这个方法生成的 *drawable* 里一样会生成一个 *bitmap* 对象。因为 *bitmap* 是通过 *bitmapfactory* 生成的，有一部分要调用*C*库所以需要开辟一部分 *native* 本地内存空间以及一部分 *JVM* 的内存空间。而 *native* 本地内存空间是*C* 开辟的，*JVM*

的 *GC* 垃圾回收器是回收不了这部分空间的，这个时候如果频繁的调用*setImageResource* 且没有手动调 *recycle native* 的内存空间很难被释放掉。

7）监听器没有注销造成的内存泄漏

- 内存泄漏原因：register listener 等监听器后，未及时 unRegister 监听器
- 解决方案：register listener 等监听器后，需要确保及时 unRegister 监听器

8）第三方库使用不当造成的内存泄漏

- 内存泄漏原因：对于 EventBus，RxJava 等一些第三方开源框架的使用，若是 Activity 销毁之前没有进行解除订阅会导致内存泄漏；
- 解决方案：Activity 销毁之前及时进行解除订阅

9）动画使用不当造成的内存泄漏（示例：BGS-22579）

- 内存泄漏原因：在属性动画中有一类无限循环动画，如果在 Activity 中播放这类动画并且在 onDestroy() 中没有去停止动画，那么动画会一直播放下去，这时候 Activity 会被 View 所持有，从而导致 Activity 无法被释放。另一种是使用帧动画，通过 BitmapFactory 来创建 bitmap，未显式调用 bitmap.recycle() 且未复用 bitmap 导致 native 内存无法正确释放。
- 解决方案：
    - 在 onDestroy() 方法中去调用 objectAnimator.cancel() 来停止动画。
    - 复用 bitmap 或者调用 bitmap.recycle() 及时释放内存。

10）WebView 使用不当造成的内存泄露

- 内存泄漏原因：WebView 内存泄露的主要原因是引用了 Activity/Fragment 的 Context，Activity/Fragment 无法被即时释放
- 解决方案：在代码中动态生成 WebView，参数 Context 引用全局的 ApplicationContext，Activity/Fragment 界面销毁时，将 WebView 从其父控件中移除，让 WebView 停止加载页面并释放或者可以为 WebView 开启一个独立的进程，使用 AIDL 与应用的主进程进行通信，WebView 所在的进程可以根据业务的需要选择合适的时机进行销毁，达到正常释放内存的目的

11）单例内存泄露

- 内存泄漏原因：在单例模式中，单例对象的生命周期与应用的生命周期相同，如果传入的 Context 是 Activity 的话，在 Activity 退出的时候，单例对象持有了 Activity 的引用，导致 Activity不能被回收；解决方案：传入 getApplicationContext() 可以解决这个问题。
-

#### 3.2.2.3 MAT 分析内存泄漏

检测出内存泄漏时，可以先 dump 内存 hprof，再用 Memory Analyzer (MAT) 分析内存泄漏的原因。

1）dump 内存 hprof。以 dump launcher3 的内存 hprof 为例：

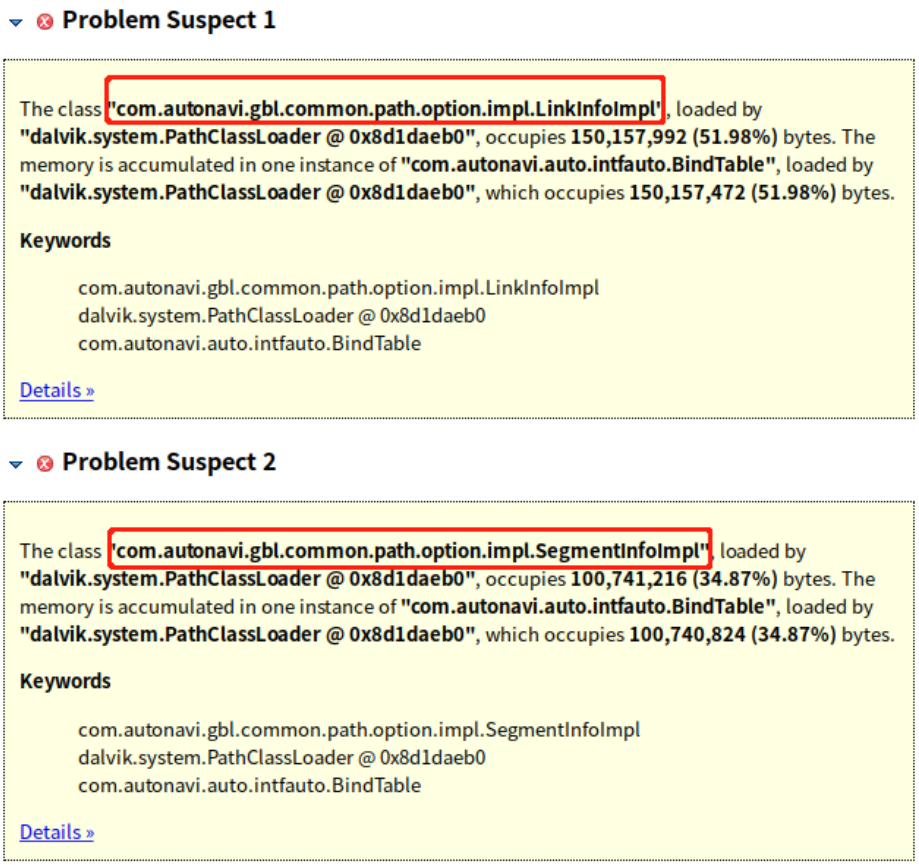
```
adb shell
am dumpheap com.android.launcher3 /data/local/tmp/launcher3.hprof
adb pull /data/local/tmp/launcher3.hprof .
```

2）android sdk hprof-conf 转换到 MAT 能打开的格式。以步骤1中得到的 launcher3 的内存 hprof 为例：

```
cd <android-sdk目录>
cd platform-tools
./hprof-conv </hprof目录/launcher3.hprof /hprof目录/launcher3_conv.hprof
```

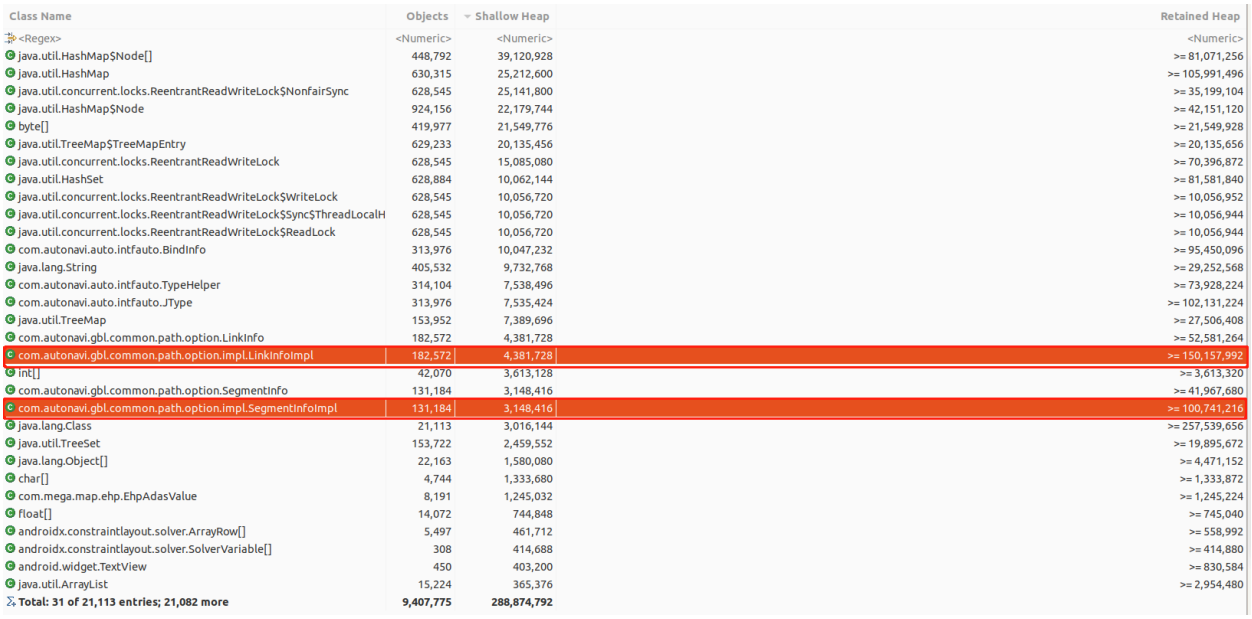
3）MAT 分析转换后的 hprof。以 BigSur 上导航内存泄漏的 BGS-13027 为例：

1. 打开步骤2中转换后的hprof,选Leak Suspects：




从上面的 Problem Suspect 中，可以看出内存中的 LinkInfoImpl，SegmentInfoImpl 等对象没有释放，可能引起内存泄漏。

2. MAT 中选 Histogram，会显示内存中各 class 的对象个数，以及 Shallow Size，Retained Size 等



| Class Name | Objects | Shallow Heap | Retained Heap |
|---|---|---|---|
| <Regex> | <Numeric> | <Numeric> | <Numeric> |
| java.util.HashMap$Node[] | 448,792 | 39,120,928 | >= 81,071,256 |
| java.util.HashMap | 630,315 | 25,212,600 | >= 105,991,496 |
| java.util.concurrent.locks.ReentrantReadWriteLock$NonfairSync | 628,545 | 25,141,800 | >= 35,199,104 |
| java.util.HashMap$Node | 924,156 | 22,179,744 | >= 42,151,120 |
| byte[] | 419,977 | 21,549,776 | >= 21,549,928 |
| java.util.TreeMap$TreeMapEntry | 629,233 | 20,135,456 | >= 20,135,656 |
| java.util.concurrent.locks.ReentrantReadWriteLock | 628,545 | 15,085,080 | >= 70,396,872 |
| java.util.HashSet | 628,884 | 10,062,144 | >= 81,581,840 |
| java.util.concurrent.locks.ReentrantReadWriteLock$WriteLock | 628,545 | 10,056,720 | >= 10,056,952 |
| java.util.concurrent.locks.ReentrantReadWriteLock$Sync$ThreadLocalH | 628,545 | 10,056,720 | >= 10,056,944 |
| java.util.concurrent.locks.ReentrantReadWriteLock$ReadLock | 628,545 | 10,056,720 | >= 10,056,944 |
| com.autonavi.auto.intfauto.BindInfo | 313,976 | 10,047,232 | >= 95,450,096 |
| java.lang.String | 405,532 | 9,732,768 | >= 29,252,568 |
| com.autonavi.auto.intfauto.TypeHelper | 314,104 | 7,538,496 | >= 73,928,224 |
| com.autonavi.auto.intfauto.JType | 313,976 | 7,535,424 | >= 102,131,224 |
| java.util.TreeMap | 153,952 | 7,389,696 | >= 27,506,408 |
| com.autonavi.gbl.common.path.option.LinkInfo | 182,572 | 4,381,728 | >= 52,581,264 |
| com.autonavi.gbl.common.path.option.impl.LinkInfoImpl | 182,572 | 4,381,728 | >= 150,157,992 |
| int[] | 42,070 | 3,613,128 | >= 3,613,320 |
| com.autonavi.gbl.common.path.option.SegmentInfo | 131,184 | 3,148,416 | >= 41,967,680 |
| com.autonavi.gbl.common.path.option.impl.SegmentInfoImpl | 131,184 | 3,148,416 | >= 100,741,216 |
| java.lang.Class | 21,113 | 3,016,144 | >= 257,539,656 |
| java.util.TreeSet | 153,722 | 2,459,552 | >= 19,895,672 |
| java.lang.Object[] | 22,163 | 1,580,080 | >= 4,471,152 |
| char[] | 4,744 | 1,333,680 | >= 1,333,872 |
| com.mega.map.ehp.EhpAdasValue | 8,191 | 1,245,032 | >= 1,245,224 |
| float[] | 14,072 | 744,848 | >= 745,040 |
| androidx.constraintlayout.solver.ArrayRow[] | 5,497 | 461,712 | >= 558,992 |
| androidx.constraintlayout.solver.SolverVariable[] | 308 | 414,688 | >= 414,880 |
| android.widget.TextView | 450 | 403,200 | >= 830,584 |
| java.util.ArrayList | 15,224 | 365,376 | >= 2,954,480 |
| **Total: 31 of 21,113 entries; 21,082 more** | **9,407,775** | **288,874,792** | |

从 Histogram 视图可以看出，LinkInfoImpl 对象有 182572 个，SegmentInfoImpl 对象有 131184个，确认这些对象没有释放，引起内存泄漏。

解决方案：高德的 pathInfo，linkInfo，segmentInfo 使用后都需要 delete。

# 4 稳定性优化

稳定性优化主要包括 Crash 和 ANR 优化。Crash 又包括 Java Crash 和 Native Crash。

## 4.1 Crash 问题处理

### 4.1.1 Java Crash

Java 的 crash 比较简单，出现 crash 时在 Logcat 中搜 fatal,会看到 crash 时进程的堆栈，在堆栈中会指出 crash 时的方法调用关系和 crash reason，crash 的 Java类和行数等，通过这些信息基本上能很快定位到 crash 的原因。

### 4.1.2 Native Crash

出现 native crash 时，会保存一个 tombstone 文件到 /data/tombstones 目录，通过 addr2line 或 ndk-stack 工具或 AOSP 中的 stack 脚本，分析这个 tomestone 文件中的 crash 堆栈地址，也能定位到 crash 时对应的 cpp 文件的行数，再结合对应的 cpp 文件中的逻辑，进一步定位 native crash 的原因

AOSP 中的 stack 脚本分析 tomstone 文件方法：

1）下载 tomestone 文件和对应的 symbol 目录到本地

2）在 AOSP 根目录执行（以 monza 项目为例）

```
. build/envsetup
lunch monza-userdebug
cd aosp/development/scripts
./stack --syms /<android_symbols目录>/android_symbols/symbols/ < /<tomestone文件目录>/tombstone_00  > /<result.txt文件目录>/result.txt
```

分析解析得到的 result.txt。

### 4.1.3 Crash 在线监控

通过集成第三方 crash 监控 SDK，如友盟移动统计 SDK 等，可以在线监控对应 App 的 crash，并对 crash 的 log 分析 crash 的原因，优化 App 的稳定性

## 4.2 ANR 优化

### 4.2.1 ANR 原因

Android 系统中，ActivityManagerService (AMS) 和 WindowManagerService (WMS) 会检测 App 的响应时间。如果 App 在特定时间内无法响应屏幕触摸或键盘输入，或者特定事件没有处理完成，就会出现 ANR。

常见 ANR 原因：

- 主线程阻塞或耗时操作，如：复杂的 Layout，庞大的 for 循环，IO，网络请求等
- 主线程被子线程同步锁 block
- 主线程被 Binder 对端 block
- Binder 被占满导致主线程无法和 SystemServer 通信
- CPU 满负荷，I/O 阻塞
- 内存泄漏引起内存过高
- 四大组件 ANR

### 4.2.2 ANR 类型

- Service Timeout：比如前台服务在 20s 内未执行完成
- BroadcastQueue Timeout：比如前台广播在 10s 内未执行完成
- ContentProvider Timeout：内容提供者，publish() 在 10s 内未处理完成

- InputDispatching Timeout：输入事件分发超时 5s，包括按键和 touch 事件

#### 4.2.3 ANR 相关日志

- logcat
- data/anr/traces.txt
- CPU 数据
- 内存 PSS/USS 等数据

#### 4.2.4 ANR 分析方法

1. 通过 trace log 确定发生 ANR 的进程 pid，包名，ANR 时间点等信息
2. 在 trace log 中查看 main 主线程状态和堆栈
3. logcat 中通过 AnrManager 相关 log，查看 ANR 的 reason，CPU等信息
4. logcat 中查看出现 ANR 的时间点附近主线程执行逻辑
5. 结合对应进程的主线程执行逻辑相关业务代码，进一步分析 ANR 的原因

#### 4.2.5 ANR 解决方案

1. UI 主线程尽量只做跟 UI 相关的工作，尽量在主线程避免耗时操作；
2. 耗时操作 (如：数据库操作，I/O，网络请求或别的有可能阻碍UI线程的操作) 放入子线程处理；
3. Activity 的关键生命周期尽量少做创建操作。

# 5 卡顿优化

## 5.1 卡顿原因

### 5.1.1 卡顿原理

人眼能感知到的流畅的画面更新帧率是 60fps，即应用必须在约 16ms 内完成屏幕刷新的全部逻辑操作，否则就会发生卡顿。在 vsync 信号（垂直同步脉冲）约16ms触发一次作用下，CPU/GPU/Display 协同处理，显示帧的画面。UI 主线程中处理一帧的任务时间必须小于 16ms，如果 UI 主线程中处理时间长，就可能导致跳过帧的渲染，也就是导致界面看起来不流畅和卡顿。

### 5.1.2 卡顿原因

从系统层面看：

1. SurfaceFlinger 主线程耗时：SurfaceFlinger 负责 Surface 的合成，一旦 SurfaceFlinger 主线程调用超时，就会产生掉帧；
2. 后台活动进程太多导致系统繁忙：dumpsys cpuinfo 可以查看一段时间内 CPU 的使用情况；
3. 主线程调度不到，处于 Runnable 状态；
4. System锁：system_server 的 AMS 锁和 WMS 锁，在系统异常时，系统的关键任务都被阻塞，等待锁的释放，这时候如果有 App 发来的 Binder 请求带锁，那么也会进入等待状态，这时候 App 就会产生卡顿问题；
5. Layer 过多导致 SurfaceFlinger Layer Compute 耗时。

从应用层面看：主线程执行 Input，Animation， Measure，Layout，Draw，decodeBitmap 等操作超时都会导致卡顿。

## 5.2 卡顿检测

1）使用dumpsys gfxinfo

```
adb shell dumpsys gfxinfo [PACKAGE_NAME]
```

比如：在 monza 上：

```
# adb shell dumpsys gfxinfo com.voyah.os.carlauncher
Applications Graphics Acceleration Info:
Uptime: 217910748 Realtime: 217910748

** Graphics info for pid 2458 [com.voyah.os.carlauncher] **
Stats since: 185593924685554ns
Total frames rendered: 1174                          # 本次dump搜集了1174帧的信息
Janky frames: 28 (2.39%)                             # 出现卡顿的帧数有28帧，占2.39%
50th percentile: 5ms
90th percentile: 6ms
95th percentile: 7ms
99th percentile: 38ms
Number Missed Vsync: 8                               # 垂直同步失败的帧
Number High input latency: 11                        # 因UI线程上的工作导致超时的帧数
Number Slow UI thread: 12
Number Slow bitmap uploads: 0                        # 因bitmap的加载耗时的帧数
Number Slow issue draw commands: 11                  # 因绘制导致耗时的帧数
```

2）使用 systrace

```
python systrace.py [options] [categories]
```

比如：

```
python systrace.py -o trace.html sched freq idle am wm gfx view  binder_driver
input res
```

执行此命令后生成 trace.html，用 Chrome 浏览器打开，即可查看 CPU 活动，VSync 事件，显示帧等，检查界面帧和提醒。渲染时间超过 16.6ms 的帧会以黄色或红色帧圆圈表示

3）Android Studio CPU Profiler

CPU Profiler可以查看主线程中，每个方法的耗时情况，以及每个方法的调用栈，可以很方便的分析卡顿产生的原因，以及定位到具体的代码方法。

## 5.3 卡顿优化方案

1）布局优化

比如：使用 ConstraintLayout 代替 LinearLayout、RelativeLayout，减少 Layout 嵌套层级，ViewStub 标签，include 标签，merge 标签等优化 Layout 布局.

2）减少主线程耗时操作

主线程中不要直接操作数据库，主线程中不要进行复杂的数据解析操作，不要在 Activity 的 onResume() 和 onCreate() 中进行耗时操作，比如大量的计算.

3）对象分配和回收优化

比如：减少小对象的频繁分配和回收操作等

4）减少过度绘制

5）列表优化

比如：用 DiffUtil 和 notifyItemDataSetChanged 对 RecyclerView 进行局部更新

文档 **Review** 反馈意见（**20220715 - li.zhuang**）：这篇开发指导文档质量非常高，内容简单明了，没有废话。稍微缺少一些相关别背景介绍，我找了些 link 链接在文档里面。文中分享的方法，在实际工程中的实施，不能只靠大家自觉性，需要在工具上和流程上去落实。因此，在这部分Cores工作正式结束前，还有几个后续需要完善的地方。性能优化相关工作，从工程管理角度分为下面几个层次：

1. 经验分享层面 – 持续进行
    a. 如：本文当中的内容，在后续工作中可以持续更新补充，提供更多例子；
    b. 组织一次关于本文内容的全公司技术分享，并且要求HR协助录像，作为IFS团队新人入职培训内容的一部分。
2. 测试流程层面 – 尽快完成、给出时间表
    a. 本文一些相关检查可以进入到工程师提交代码之前自测环节。对于这些可以自测，能否有一个简单的脚本或者特别容易用的工具，做一些一句命令行脚本就能完成的基础检查和报告内容，作为提交前检查。
    b. 对内存/CPU占用等、Crash、卡顿等等相关的检查，进入测试的Matrix系统检查流程，制定。需要跟测试确定检查和测试频率。这个目的是为了如果程序写坏了，可以及时发现。
3. 调试方法技巧层面 – 尽快完成、给出时间表
    a. 目前 Matrix 测试经常会产生很多 Crash，TOMESTONE，ANR，放很长时间没有人处理。需要形成一个技巧性的文档，收到这种JIRA ticket，按步骤应该怎样去调试。
    b. Review 一次 Matrix 压测的检测和去重机制，形成一个对 Matrix 压测的 Review 报告，确认当前 Matrix 的误报和漏报情况。
4. 发版流程层面 – 尽快完成、给出时间表
    a. 本文中提到的优化和检查，正式发用户版本前，哪些必须进行，给出一个 check list；
    b. 以 BigSur 为例，做一个性能优化报告（做了哪些工作，每一项工作的优化效果如何），作为以后项目正式发 SOP 版本的参考；可以参考和修改模板
    Android System App 启动优化 - 模板 Android 内存优化报告 - 模板
    c. Check list 给出后，我会要求质量部门放到正式用户版本发版流程中。