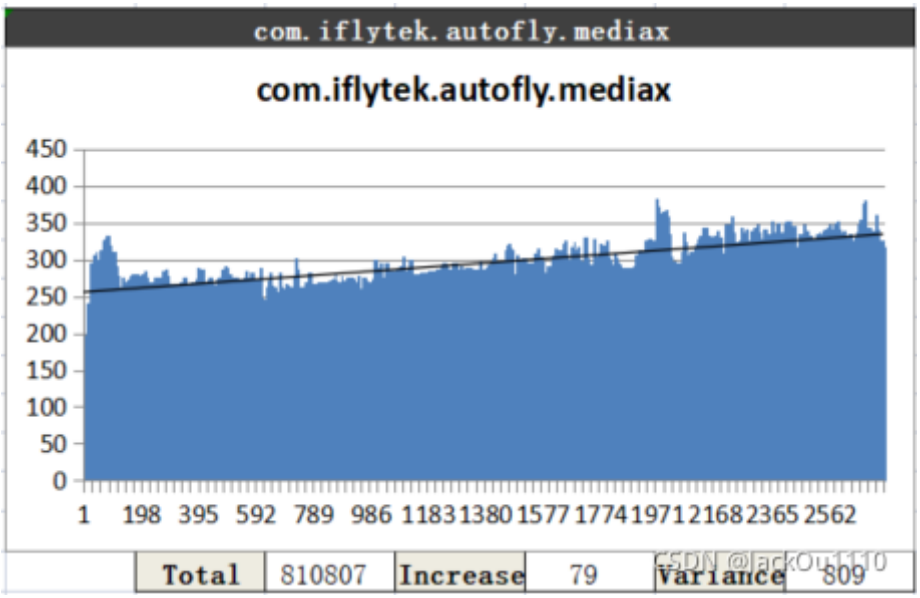
> title: 聚媒体内存问题分析与优化建议
>
> author: 欧杰 莫国权

0.前言

从《GOS项目车机整体内存性能问题分析报告》中可以看到媒体应用是比较消耗内存的应用，并且确实存在一定内存问题，因此此报告主要针对聚媒体应用的宏观测出的内存问题，细节分析存在问题点和优化方案。

从报告中取出聚媒体的内存现状如下：



最大内存	最小内存	平均内存	内存抖动状态	是否内存泄露	是否阶梯内存
383M	145M	295M	存在	存在	存在

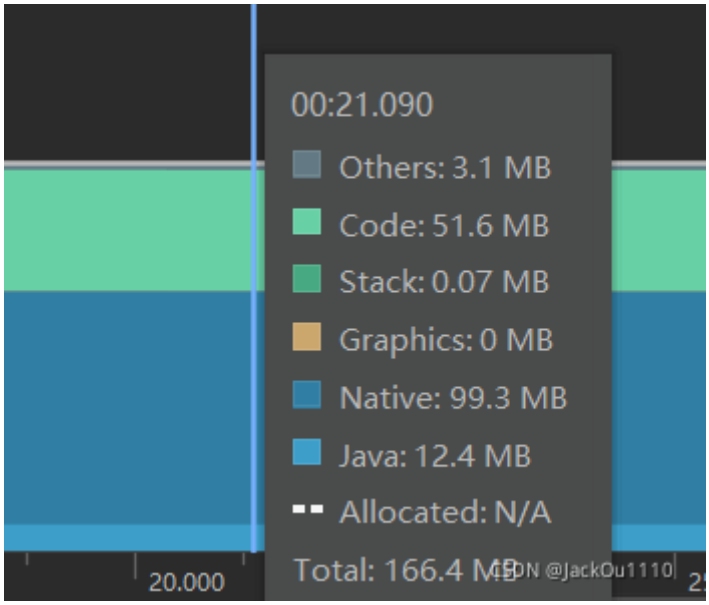
因此，我们从如下三个方面来看看聚媒体如何优化：

- 大内存分配
- 内存抖动
- 内存泄露

1.大内存分配与内存抖动

1.1 APP静止状态内存

静止状态内存可以粗略理解成媒体应用非界面运行所消耗的内存，也可以理解成运行后台服务消耗的内存。



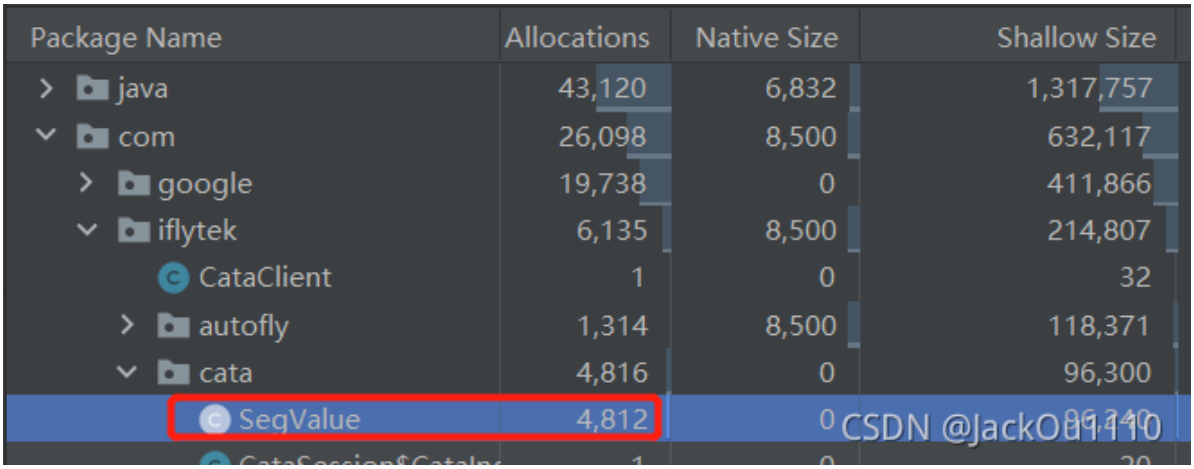
媒体应用静止情况下占用内存166.4M，其中Native消耗99.3M，Code消耗51.6M，java消耗12.4M。

- **Native**：从 C 或 C++ 代码分配的对象的内存。

  即使您的应用中不使用 C++，您也可能会看到此处使用了一些原生内存，因为即使您编写的代码采用 Java 或 Kotlin 语言，Android 框架仍使用原生内存代表您处理各种任务，如处理图像资源和其他图形。
- **Code**：您的应用用于处理代码和资源（如 dex 字节码、经过优化或编译的 dex 代码、.so 库和字体）的内存。
- **Java**：从 Java 或 Kotlin 代码分配的对象的内存。

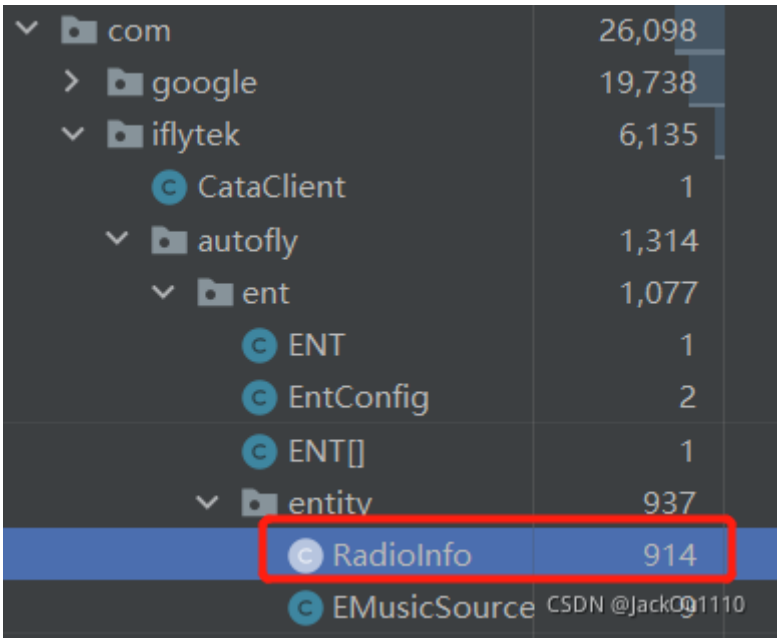
如下对象需要FO调查一下是否分配合理，针对多次调用的场景是否可以复用。

1.1.1 SegValue对象



在静止状态媒体应用会创建4812个SegValue对象，需要FO看一下是否用到，我在代码中没有grep到对应的引用。

1.1.2 RadioInfo对象



在静止状态媒体应用当扫描到电台频段的时候，会创建一个RadioInfo()对象，从图中可以看出，在此轮扫描中创建了914个对象。此处请使用 对象池 来添加对象。

PS: 代码中还有创建RadioInfo()对象的地方，请调查一下使用对象池。

优化建议：

示例如下，**经过如下修复之后，可以减少大对象创建所消耗的内存和减少内存抖动。**

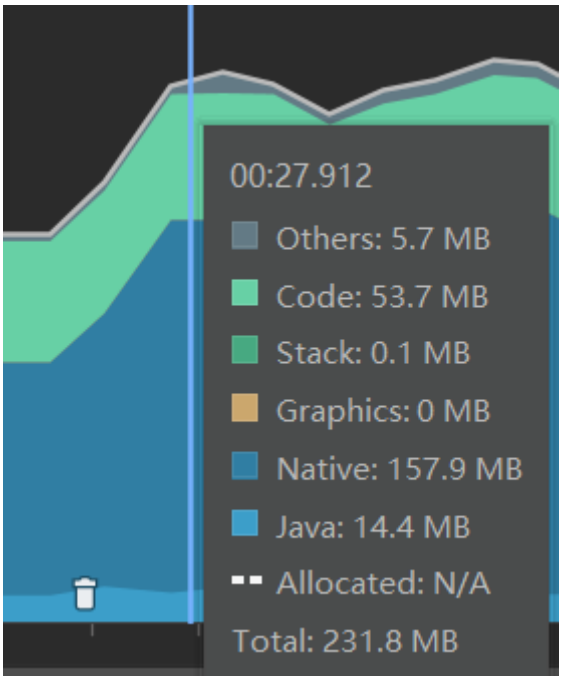
```
//
mediax\common\src\main\java\com\gxatek\mediax\common\usecase\RadioScanUseCase.ja
va
@Override
public void onScanFrequency(int frequency, boolean valid) {
    ......

        RadioInfo findRadioInfo = new RadioInfo();   // 此处从对象池中获取一个对象
        // RadioInfo findRadioInfo = mPool.acquire();
        findRadioInfo.setFrequency(String.valueOf(frequency));

        ......
        // 对对象进行操作
        ......

        if (!validRadioInfosPart.contains(findRadioInfo)) {
            validRadioInfosPart.add(Math.max(searchIndex, 0), findRadioInfo); //
把对象加入列表
            // 释放对象
            // mPool.release(findRadioInfo);
        }

        getUseCaseCallback().onSuccess(
                new ResponseValue<>(new Pair<>(validRadioInfosPart,
searchIndex),
                        new ResponseStatus(ResponseCode.DATA_IS_VALID, true)));
    }
}
```

1.1.3 Gson的TypeToken

代码中大量用到Gson解析json字段，在解析之前都创建了一个TypeToken对象，请FO查看该对象是否可以缓存下来复用。

```
// 代码举例
//
mediax\common\src\main\java\com\gxatek\mediax\common\usecase\ResumePlayRadioUseCase.java
switch (requestValues.radioType) {
    case RadioServiceCons.RadioType.FM:
        // 如下两个TypeToken的Type是一样的，请复用Type
        mHistoryRadioList =
GsonUtil.fromJson(SPUtil.getInstance().getString(Configs.HISTORY_FM_RADIO_LIST),
new TypeToken<List<RadioInfo>>(){}.getType());
        break;
    case RadioServiceCons.RadioType.AM:
        // 如下两个TypeToken的Type是一样的，请复用Type
        mHistoryRadioList =
GsonUtil.fromJson(SPUtil.getInstance().getString(Configs.HISTORY_AM_RADIO_LIST),
new TypeToken<List<RadioInfo>>(){}.getType());
        break;
}
```

1.2 媒体库主界面



	Total	Native	Code	Java
静止状态	166.4M	99.3M	51.6M	12.4M
主界面	231.8M	157.9M	53.7M	14.4M

从静止状态到进入主界面，内存消耗**增加65.4M**，其中主要是Native消耗了内存，**占用58.6M**

因此，从宏观上，主界面增长的内存主要是Native消耗了，根据经验，主要是加载Bitmap消耗了，接下来，我们来调查Bitmap消耗的内存情况。

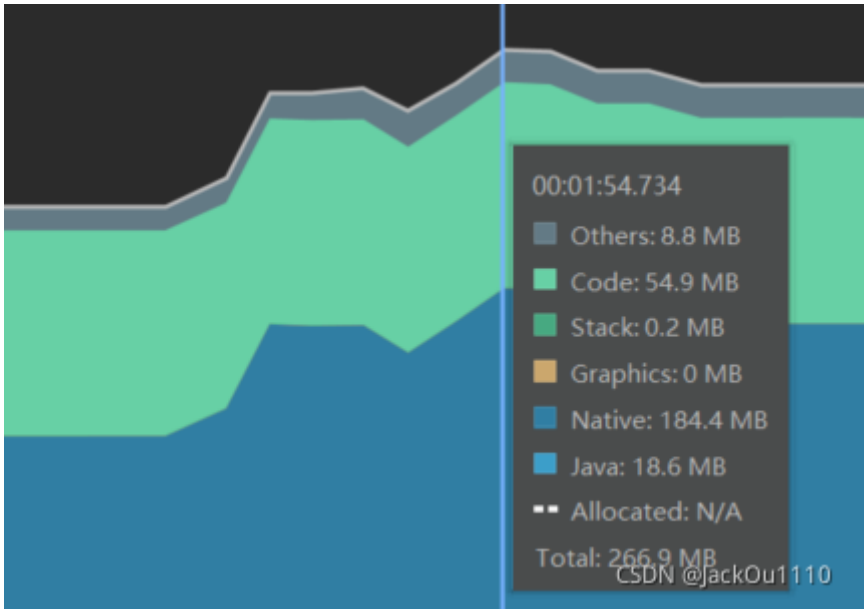
可以从图中看到主界面加载了38个Bitmap对象，消耗24M内存，逆向推导，查看bitmap加载的方法。从代码中可以看到主界面的图片是使用Glide加载的，代码如下：

```
//
mediax\common\src\main\java\com\gxatek\mediax\common\binding_adapter\ImageBindingAdapter.java
Glide.with(view)
    .load(imageUrl)
    .diskCacheStrategy(DiskCacheStrategy.RESOURCE)
    .error(error_resId)
    .placeholder(place_resId)
    .error(error_resId)
    .apply(options)
    .into(view);
```

Glide是一个非常优秀的图片加载框架，但是使用Glide来加载图片会依赖于云端给的图片大小，如果云端给的图片特别大，而我们使用的空间比较小，就会出现过度加载的情况，因此推荐在使用Glide加载图片的时候，使用`override`方法来对图片进行按照空间的尺寸压缩。示例如下：

```
Glide.with(view)
    .load(imageUrl)
    .diskCacheStrategy(DiskCacheStrategy.RESOURCE)
    .error(error_resId)
    .placeholder(place_resId)
    .error(error_resId)
    .override(200, 200)    // 控件大小为200 * 200，加载图片大小自己控制，不受制于云端。
    .apply(options)
    .into(view);
```

1.3 酷我音乐界面

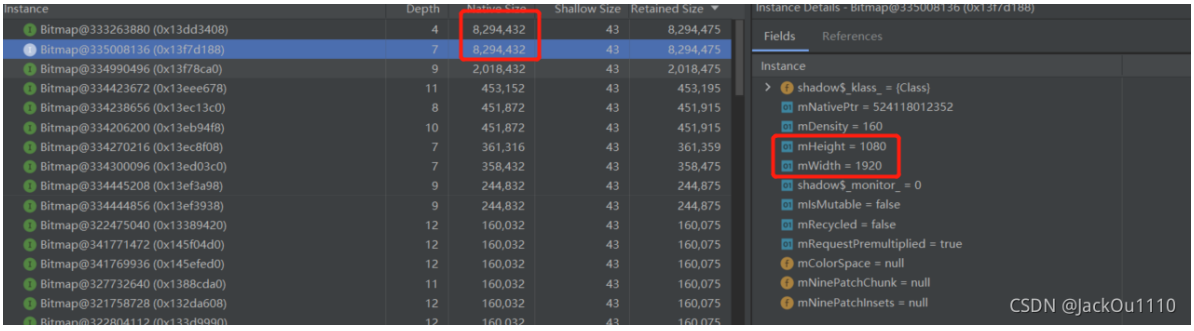


	Total	Native	Code	Java
主界面	224.9M	150.7M	54.9M	13.1M
酷我音乐界面	257.7M	174.9M	54.9M	18.9M

从主界面进入酷我音乐界面，内存消耗**增加32.8M**，其中主要是Native消耗了内存，**占用24.2M。**

酷我音乐界面创建了124个Bitmap对象，占用34M内存。

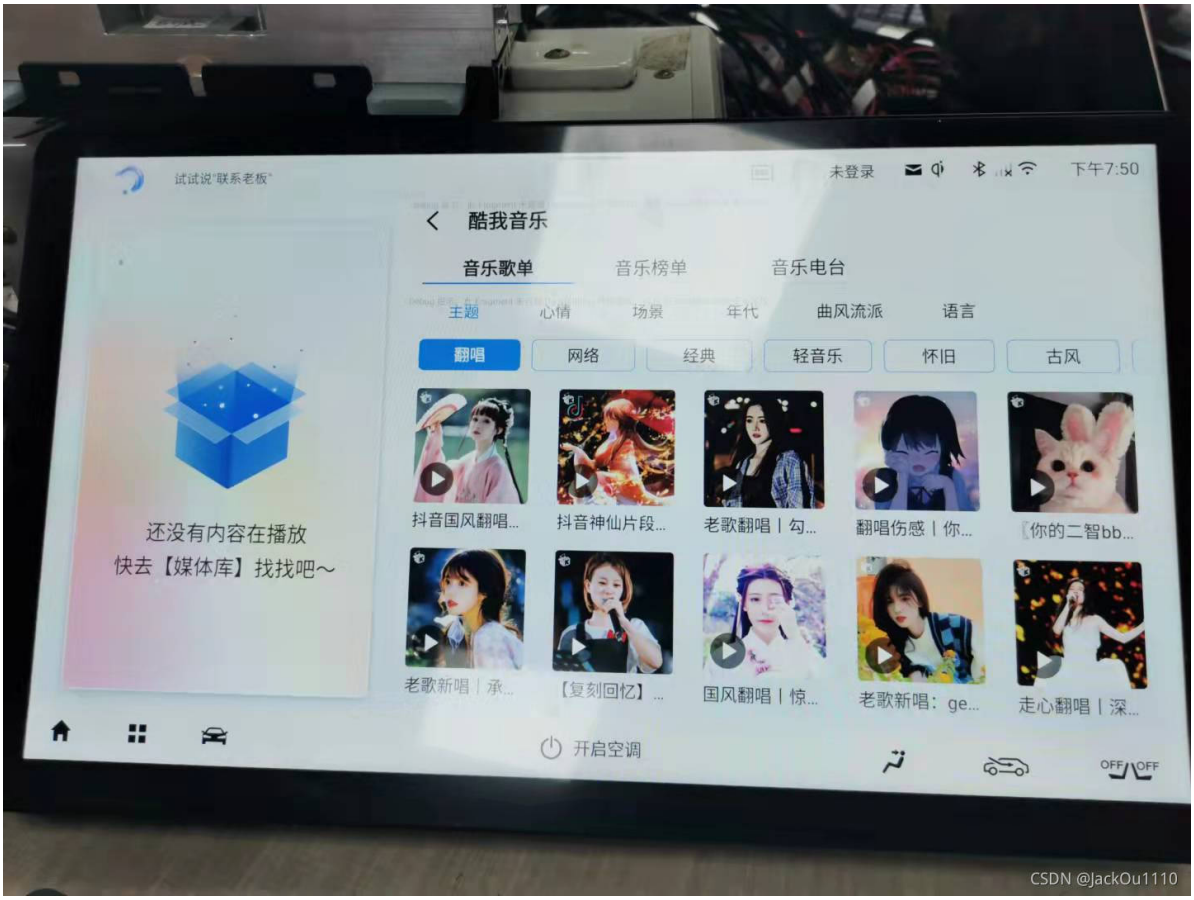
1.3.1 背景图片重复加载



图中可以看到有两个1080*1920尺寸的图片加载，此处怀疑是**该界面有两层背景图片！**一张图片暂用8M左右内存。

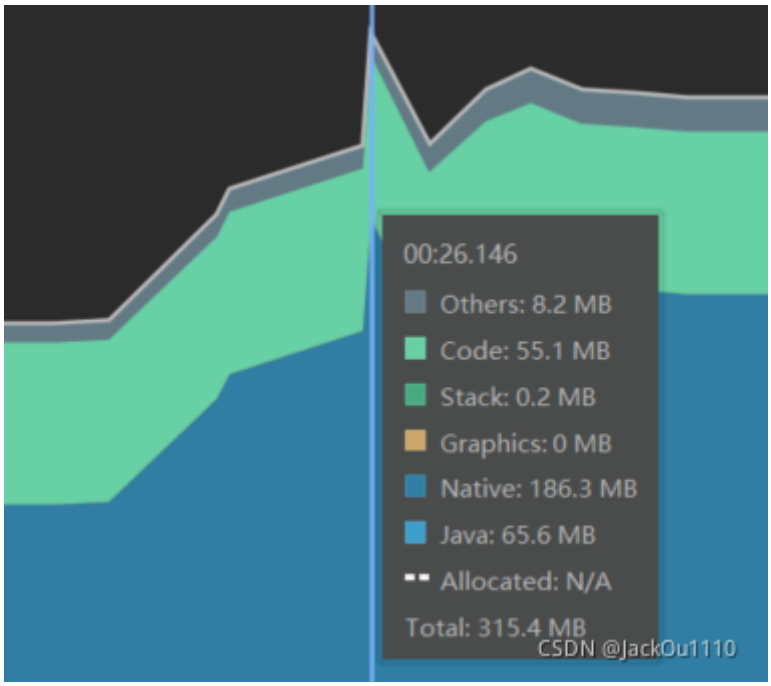
PS：主界面，酷我音乐界面，喜马拉雅界面都需要看一下是否有重复加载背景图片的情况。

1.3.2 过度加载bitmap



从图中看到RecycleView中**不滑动只显示10个图片，滑动也最多显示15个图片**，实际上该界面加载了**124个Bitmap**，需要FO对该RecycleView加载的内容数做限制。

1.4 喜马拉雅听界面

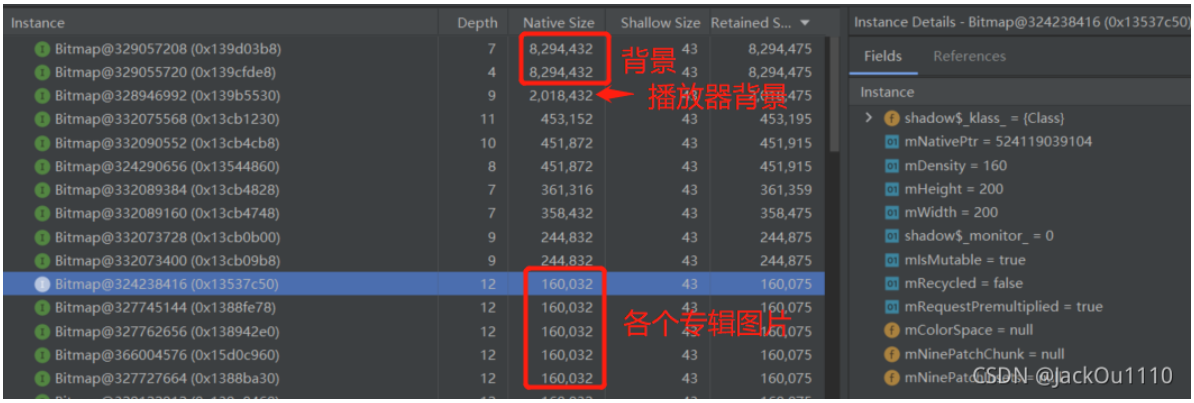


	Total	Native	Code	Java
主界面	217.5M	141.4M	54.9M	14.4M
酷我音乐界面	315.4M	186.3M	55.1M	65.6M

从主界面进入喜马拉雅FM界面，内存消耗**增加97.9M**，其中主要是Native消耗了内存，**占用44.9M。**

喜马拉雅FM界面创建Bitmap对象91个，消耗内存31.4M。

1.4.1 Bitmap内存占用问题



从上图中可以看出问题和酷我音乐界面一样

- **背景加载了两次**
- **节目图片加载过多，需要按需加载。**

1.4.2 自定义图片加载工具

从代码中可以找到两个自定义的图片加载工具，分别是`BitmapUtil`和`FrameBitmapUtil`，在这两个工具中加载图片环节都有一点点小问题：**就是没有根据控件大小，对图片进行压缩加载，给进来的图片多大就加载多大，站在内存资源的角度上看，对于大图片加载在小控件上，属于资源浪费。**

```
//
mediax\localmusic\src\main\java\com\iflytek\autofly\localmusic\utils\BitmapUtil.
java
//
mediax\app\src\main\java\com\iflytek\autofly\mediax\utils\FrameBitmapUtil.java

public static Bitmap loadBitmap(String drawableResPath) {
        Bitmap frameBitmap = null;
        BufferedSource bufferedSource = null;
        try {
            final InputStream frameInputStream =
ENT.I.getContext().getAssets().open(drawableResPath);
            bufferedSource = Okio.buffer(Okio.source(frameInputStream));
            byte[] imageBytes = bufferedSource.readByteArray();
            BitmapFactory.Options options = new BitmapFactory.Options();
            options.inSampleSize = 1;// 从这两个工具类的加载图片方法中都看到直接把
inSampleSize写死设置成1，不进行缩放。
            if (isReusableBitmap) {
                options.inJustDecodeBounds = true;
                BitmapFactory.decodeByteArray(imageBytes, 0, imageBytes.length,
options);
                options.inJustDecodeBounds = false;
                addInBitmapOptions(options);
            }
            frameBitmap = BitmapFactory.decodeByteArray(imageBytes, 0,
imageBytes.length, options);
```

```
            if (isReusableBitmap) {
                reuseBitmap(frameBitmap);
            }
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (null != bufferedSource) {
                try {
                    bufferedSource.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
        return frameBitmap;
    }
```

以上方法可能在我们当前项目中不会存在问题，因为当前项目给的切图都是按照需求大小给的，在供给侧已经解决了`大图片加载在小控件的问题`。但是我还是建议**把这个方法做一点扩展，加上根据控件大小缩放图片的功能。**代码见`4.2 自定义Bitmap加载类`。

2.内存溢出调查

2.1 调查过程

在内存优化场景中，我们使用三个工具全方位的定位，解决问题。使用lint先把明显的问题查出来，然后使用leakcanary查看Activity，Fragment等内存泄露，通常Leakcanary都能将内存泄露发生的引用链打出来，而且可以动态监控内存泄漏，可以消除绝大多数内存泄漏点。对于比较麻烦不容易观察的，可以借助MAT工具，分析Incoming和Outgoing References，画引用关系图来定位泄漏点。

另外，还可以使用指令粗略观看是否有泄露。

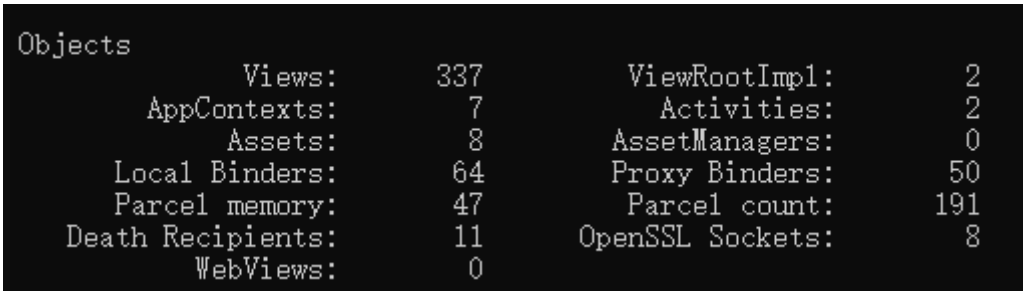
2.1.0 产生内存泄露测试手顺：

- 在不同的 Activity 状态下，先将设备从纵向旋转为横向，再将其旋转回来，这样反复旋转多次。旋转设备经常会使应用泄漏 `Activity`、`Context` 或 `View` 对象，因为系统会重新创建 `Activity`，而如果您的应用在其他地方保持对这些对象其中一个的引用，系统将无法对其进行垃圾回收。
- 在不同的 Activity 状态下，在您的应用与其他应用之间切换（导航到主屏幕，然后返回到您的应用）。

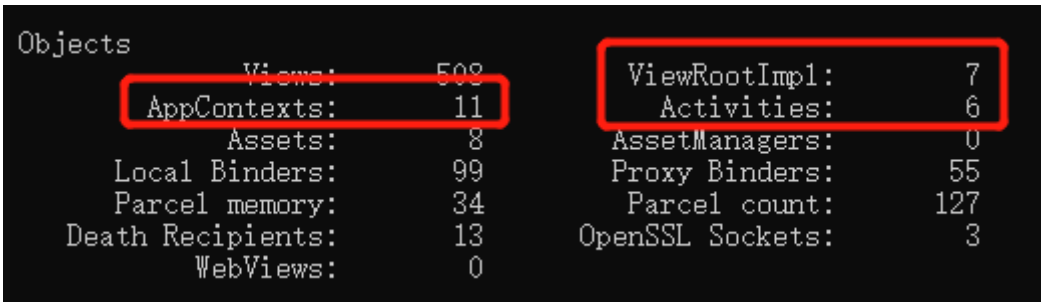
2.1.1 指令粗略查看

```
dumpsys meminfo -a <pid>
// pid为待查看应用的进程id
```

测试前：



使用2.1.0手顺猛切换测试后：



可以看出本地电台界面存在内存泄露，此方法只能大致定位是否存在泄露，调用栈需要结合leakcanary或者MAT。

2.1.2 集成lint

Android Lint 是Android自带的代码检查工具，它能帮助我们识别很多潜在的错误。对于大功能能够将明显的问题帮忙扫描出来。

> lint可以检测出如下几类问题：
>
> Correctness 不够完美的编码，比如硬编码、使用过时 API 等
> Performance 对性能有影响的编码，比如：静态引用，循环引用等
> Internationalization 国际化，直接使用汉字，没有使用资源引用等
> Security 不安全的编码，比如在 WebView 中允许使用 JavaScriptInterface 等
> Usability 可用的，有更好的替换的 比如排版、图标格式建议.png格式 等
> Accessibility 辅助选项，比如ImageView的contentDescription往往建议在属性中定义 等

针对性能问题，直接选择Performance这项，根据提示修复。

2.1.2.1 内存抖动

```
//
mediax\app\src\main\java\com\iflytek\autofly\mediax\ui\view\lyric\LrcView.java
@Override
    protected void onDraw(Canvas canvas) {
        super.onDraw(canvas);
        canvas.save();
        canvas.clipRect(new Rect(getPaddingLeft(), 0, getWidth() -
getPaddingRight(),
                Integer.MAX_VALUE), Region.Op.INTERSECT);  // 禁止在三大绘制流程中创
建对象
        ......
}
```

onMeasure()，onLayout()，onDraw()三大绘制的回调方法会大量被调用，在这三个方法中创建对象，一定会出现内存抖动，禁止在三大绘制流程中创建对象。

2.1.2.2 ImageView+TextView合并

```
//
mediax\app\src\main\res\layout\media_group_qqmusic_recommend_view_stub_layout.xml
<LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginEnd="@dimen/x40"
    android:onClick="@{()->click.replaceRecommendMusic()}"
    android:orientation="horizontal"
    android:visibility="@{showMusicReplacementButton ? View.VISIBLE :
View.GONE}"
    app:layout_constraintBottom_toBottomOf="@+id/tv_music_recommend"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="@+id/tv_music_recommend">

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/library_icon_refresh" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginStart="4dp"

 android:contentDescription="@string/visible_to_say_kw_music_recommend_change"
        android:text="@string/change_tip"
        android:textColor="@color/c_e6000000"
        android:textSize="@dimen/s28" />
</LinearLayout>
```

此布局会涉及到3个view(LinearLayout、ImageView、TextView)的加载，然后只做了`一个旋转图片和文字的展示`，此布局文件可以优化成一个view，代码参考如下：

```
<TextView
    android:id="@+id/tv_handler"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:drawableLeft="@drawable/library_icon_refresh"    // 将图片放在文字左边即
可
    android:text="换一换"
    android:textColor="@color/white"
    android:textSize="28sp"/>
```

以上优化可以将三个view优化到一个view加载，FO根据需求调整以下边界距离即可。

2.1.2.3 过度绘制

```
// mediax\app\src\main\res\layout\fragment_player_empty_new.xml
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="@dimen/x520"
    android:layout_height="match_parent"
    android:background="@mipmap/default_bg">    // 此处背景过度绘制
```

```
// mediax\app\src\main\res\layout\layout_accout_bind.xml
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/dialog_white_bg">  // 此处背景过度绘制
```

```
// mediax\app\src\main\res\layout\layout_accout_unbind.xml
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/dialog_white_bg">    // 此处背景过度绘制
```

2.1.2.4 内存泄露

```
//
mediax\app\src\main\java\com\iflytek\autofly\mediax\gvoice\soundhound\Recorder.j
ava
public class Recorder implements IIRecListener {
    /**
     * 单例类
     */
    private static Recorder mInstance;
    private Context mContext;
}
```

```
//
mediax\app\src\main\java\com\iflytek\autofly\mediax\voice\soundhound\Recorder.ja
va
public class Recorder implements IIRecListener {

    ......
    private Context mContext;
    private static Recorder mInstance;
}
```

静态单例中传入Context，可能产生内存泄露，如果代码中没有使用到，请移除该代码。

2.1.2.5 布局层级冗余

```
// mediax\app\src\main\res\layout\refresh_header.xml
```

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"   // 该布局
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:gravity="bottom" >

    <RelativeLayout          // 该布局
        android:id="@+id/listview_header_content"
        android:layout_width="fill_parent"
        android:layout_height="80dp"
        android:paddingTop="10dip">

        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:minWidth="100dip"
            android:layout_centerInParent="true"
            android:gravity="center"
            android:orientation="vertical"
            android:id="@+id/listview_header_text">

            ......
            </LinearLayout>
        </LinearLayout>

        <ImageView
            android:id="@+id/listview_header_arrow"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_centerVertical="true"
            android:layout_marginLeft="35dp"
            android:layout_marginRight="10dp"
            android:layout_toLeftOf="@+id/listview_header_text"
            android:src="@mipmap/ic_pulltorefresh_arrow" />

        <com.iflytek.autofly.mediax.ui.view.xrecyclerview.SimpleViewSwitcher
            android:id="@+id/listview_header_progressbar"
            android:layout_width="30dip"
            android:layout_height="30dip"
            android:layout_toLeftOf="@+id/listview_header_text"
            android:layout_centerVertical="true"
            android:layout_marginLeft="40dp"
            android:layout_marginRight="10dp"
            android:visibility="invisible" />
    </RelativeLayout>

</LinearLayout>
```

以上两个布局可以取消一个，要么取消LinearLayout，要么取消RelativeLayout。

2.1.2.6 移除没有使用到的布局文件和资源

此优化可以根据后期是否会用到，考虑是否删除，建议没用到的都删除，以后要用到再回退出来，此优化可以优化apk大小和内存中code大小。

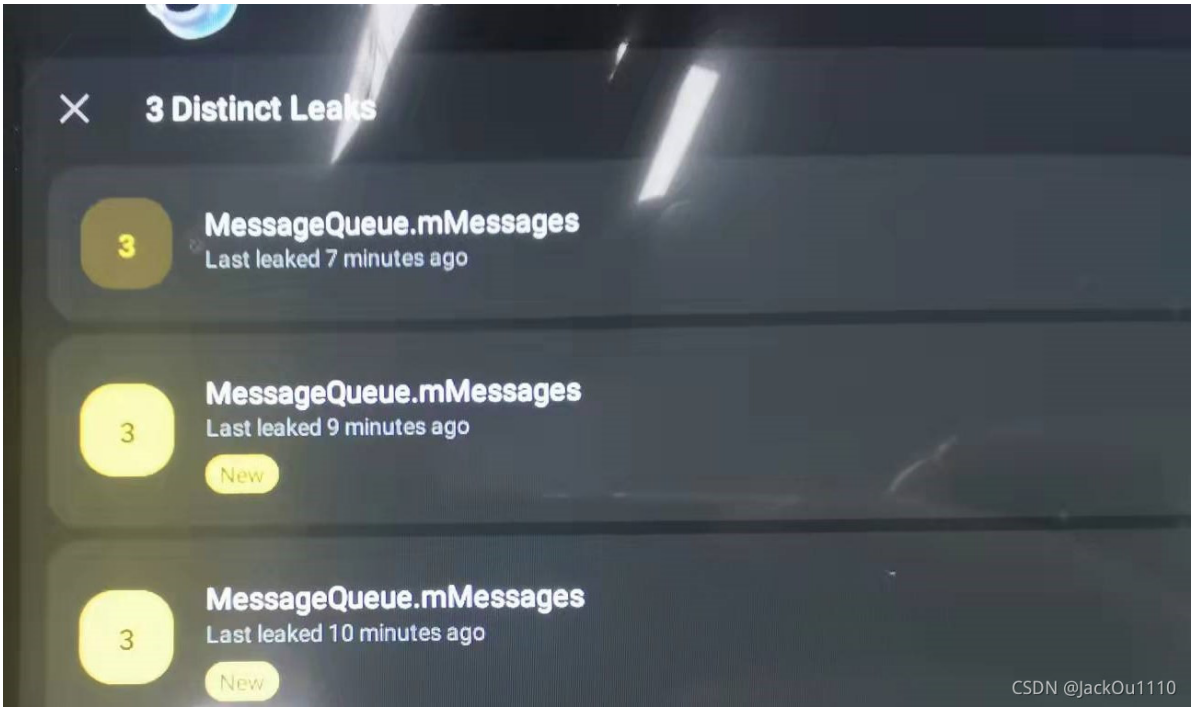
以下文件都有没有引用到的资源：

2.1.3 集成leakcanary

Leakcanary是一个检查内存泄露的工具，能够动态，大范围地锁定内存泄露问题。

```
// 在app模块的build.gradle中添加依赖，重新烧APK，使用2.1.0猛切换界面，如果有泄露会有日志和堆栈打出
implementation 'com.squareup.leakcanary:leakcanary-android:2.7'
```

2.1.3.1 Leakcanary泄露警告



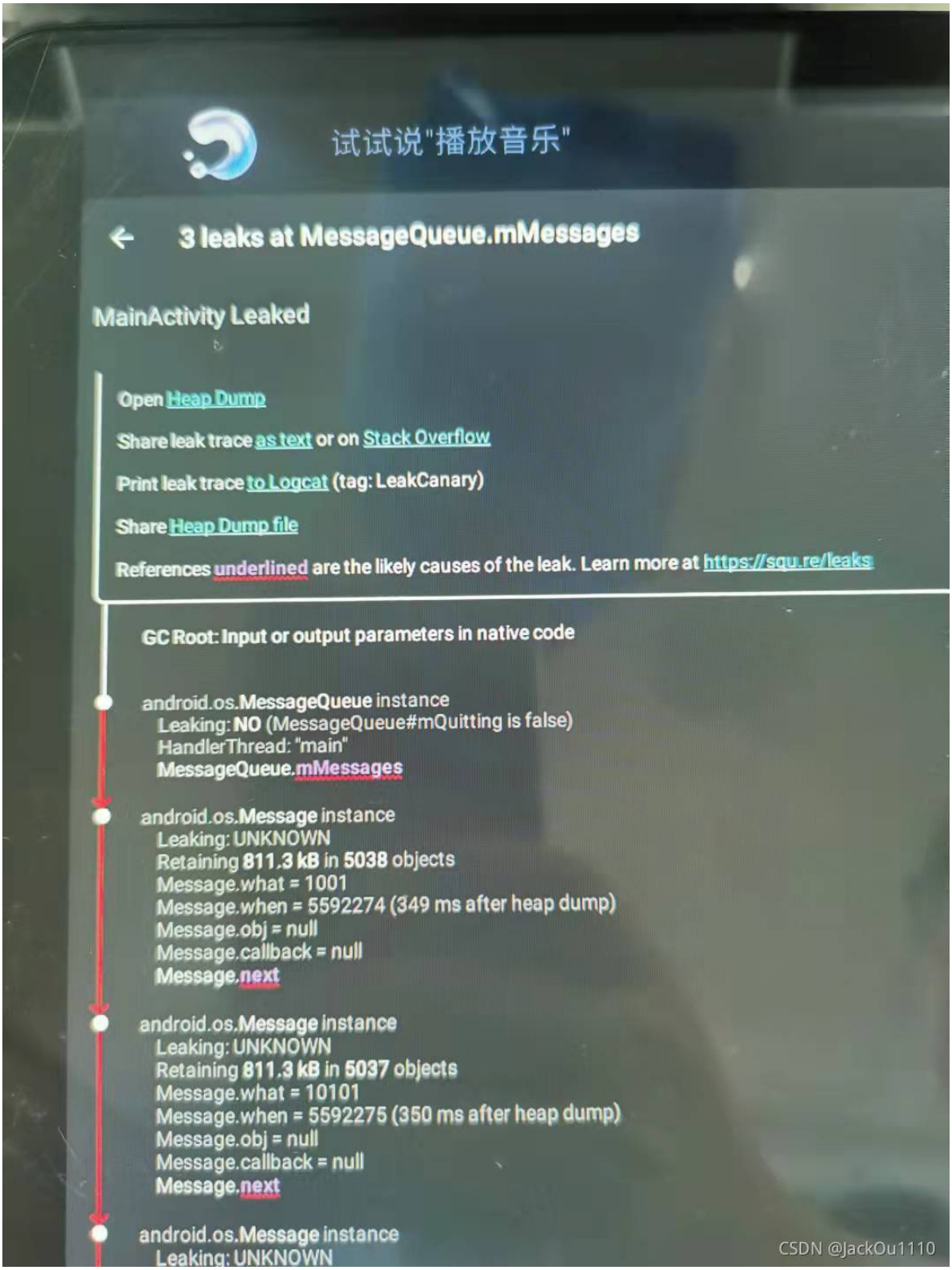
在使用`本地电台`界面，leakcanary报出了泄露问题。

2.1.3.2 泄露日志调用栈

```
!/com.iflytek.autofly.mediax D/LeakCanary: ===================================
!/com.iflytek.autofly.mediax D/LeakCanary: HEAP ANALYSIS RESULT
!/com.iflytek.autofly.mediax D/LeakCanary: ===================================
!/com.iflytek.autofly.mediax D/LeakCanary: 1 APPLICATION LEAKS
!/com.iflytek.autofly.mediax D/LeakCanary: References underlined with "~~~" are likely causes.
!/com.iflytek.autofly.mediax D/LeakCanary: Learn more at https://squ.re/leaks.
!/com.iflytek.autofly.mediax D/LeakCanary: 96705 bytes retained by leaking objects
!/com.iflytek.autofly.mediax D/LeakCanary: Displaying only 1 leak trace out of 3 with the same s
!/com.iflytek.autofly.mediax D/LeakCanary: Signature: 2b847532f5ccfccb812c61aef97098be495e91
!/com.iflytek.autofly.mediax D/LeakCanary: ┬───
!/com.iflytek.autofly.mediax D/LeakCanary: │ GC Root: Input or output parameters in native code
!/com.iflytek.autofly.mediax D/LeakCanary: │
!/com.iflytek.autofly.mediax D/LeakCanary: ├─ android.os.MessageQueue instance
!/com.iflytek.autofly.mediax D/LeakCanary: │    Leaking: NO (MessageQueue#mQuitting is false)
!/com.iflytek.autofly.mediax D/LeakCanary: │    HandlerThread: "main"
!/com.iflytek.autofly.mediax D/LeakCanary: │    ↓ MessageQueue.mMessages
!/com.iflytek.autofly.mediax D/LeakCanary: │                      ~~~~~~~~~
!/com.iflytek.autofly.mediax D/LeakCanary: ├─ android.os.Message instance
!/com.iflytek.autofly.mediax D/LeakCanary: │    Leaking: UNKNOWN
!/com.iflytek.autofly.mediax D/LeakCanary: │    Retaining 811.3 kB in 5038 objects
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.what = 1001
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.when = 5592274 (349 ms after heap dump)
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.obj = null
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.callback = null
!/com.iflytek.autofly.mediax D/LeakCanary: │    ↓ Message.next
!/com.iflytek.autofly.mediax D/LeakCanary: │              ~~~~
!/com.iflytek.autofly.mediax D/LeakCanary: ├─ android.os.Message instance
!/com.iflytek.autofly.mediax D/LeakCanary: │    Leaking: UNKNOWN
!/com.iflytek.autofly.mediax D/LeakCanary: │    Retaining 811.3 kB in 5037 objects
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.what = 10101
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.when = 5592275 (350 ms after heap dump)
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.obj = null
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.callback = null
!/com.iflytek.autofly.mediax D/LeakCanary: │    ↓ Message.next
!/com.iflytek.autofly.mediax D/LeakCanary: │              ~~~~
!/com.iflytek.autofly.mediax D/LeakCanary: ├─ android.os.Message instance
!/com.iflytek.autofly.mediax D/LeakCanary: │    Leaking: UNKNOWN
!/com.iflytek.autofly.mediax D/LeakCanary: │    Retaining 445.8 kB in 3119 objects
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.what = 10101
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.when = 5592276 (351 ms after heap dump)
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.obj = null
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.callback = null
!/com.iflytek.autofly.mediax D/LeakCanary: │    ↓ Message.next
!/com.iflytek.autofly.mediax D/LeakCanary: │              ~~~~
!/com.iflytek.autofly.mediax D/LeakCanary: ├─ android.os.Message instance
!/com.iflytek.autofly.mediax D/LeakCanary: │    Leaking: UNKNOWN
!/com.iflytek.autofly.mediax D/LeakCanary: │    Retaining 157 B in 4 objects
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.what = 10101
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.when = 5592276 (351 ms after heap dump)
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.obj = null
!/com.iflytek.autofly.mediax D/LeakCanary: │    Message.callback = null
!/com.iflytek.autofly.mediax D/LeakCanary: │    ↓ Message.target
!/com.iflytek.autofly.mediax D/LeakCanary: │              ~~~~~~
!/com.iflytek.autofly.mediax D/LeakCanary: ├─ android.os.Handler instance
!/com.iflytek.autofly.mediax D/LeakCanary: │    Leaking: UNKNOWN
!/com.iflytek.autofly.mediax D/LeakCanary: │    Retaining 37 B in 2 objects
```

下面还有内容，但是大概已经可以猜到是handler泄露场景

2.1.3.3 Leakcanary界面版调用栈



2.1.3.4 leakcanary小结

基于以上调用栈基本可以定位问题了，接下来就是查看源码，修复问题即可。但是为了演示比较难的泄露调查过程，下面会结合MAT分析细节分析引用情况。

2.1.4 使用MAT分析对象引用情况

2.1.4.1 leakcanary产生的dump日志

以上leakcanary产生的dump文件默认存在：

```
/storage/emulated/0/Download/leakcanary-com.iflytek.autofly.mediax/2021-10-26_11-21-58_522.hprof
```

2.1.4.2 自己手动打印堆栈信息

```
// 手动采集heap dump文件
public static boolean createDumpFile(Context context) {
        Log.i(TAG, "start to dump heap....");
        String LOG_PATH = "/dump.gc/";
        boolean ret = false;
        SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd_HH.mm.ssss");
        String createTime = sdf.format(new Date(System.currentTimeMillis()));
        String state = Environment.getExternalStorageState();
        if (Environment.MEDIA_MOUNTED.equals(state)) {
            File file = new
File(Environment.getExternalStorageDirectory().getPath() + LOG_PATH);
            if (!file.exists()) {
                file.mkdirs();
            }
            String hprofPath = file.getAbsolutePath();
            if (!hprofPath.endsWith("/")) {
                hprofPath += "/";
            }

            hprofPath += createTime + ".hprof";
            try {
                Debug.dumpHprofData(hprofPath);
                ret = true;
                Log.d(TAG, "createDumpFile: done!");
            } catch (IOException e) {
                e.printStackTrace();
            }
        } else {
            ret = false;
            Log.d(TAG, "NO SDCARD");
        }
        return ret;
    }
```

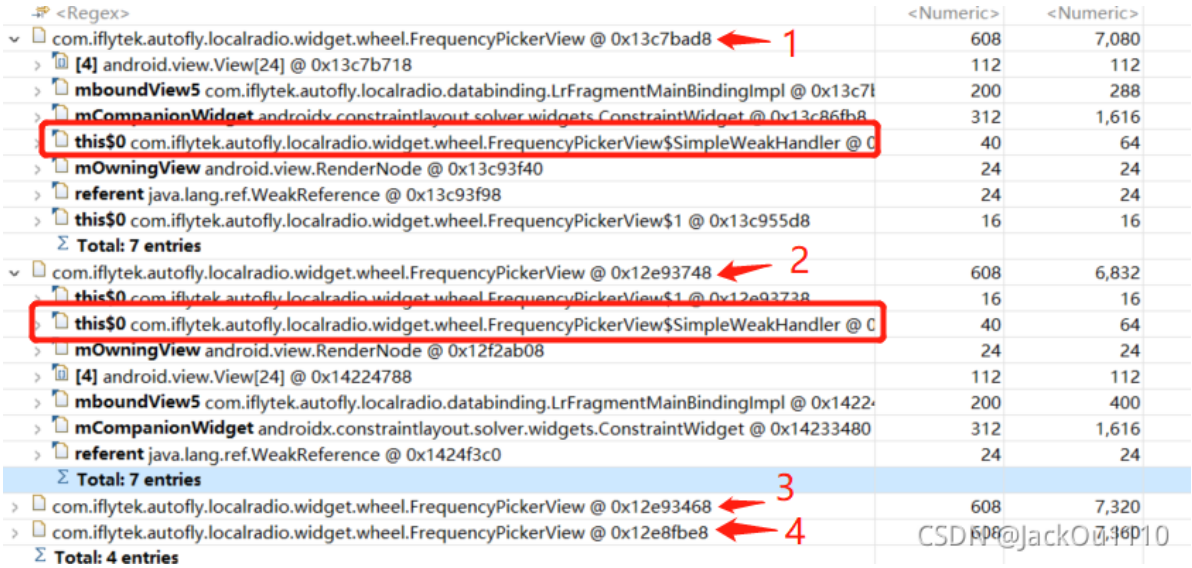
2.1.4.3 转化hropf文件

```
hprof-conv heap-original.hprof heap-converted.hprof
```

2.1.4.4 MAT分析对象无法释放

	Objects	Shallow Heap	Retained Heap
FrequencyPickerView	4	2,432	>= 35,912
FrequencyPickerView$SimpleWeakHandler	4	160	>= 800
FrequencyPickerView$1	4	64	>= 360
FrequencyPickerView$SimpleTimerTask	0	0	>= 312

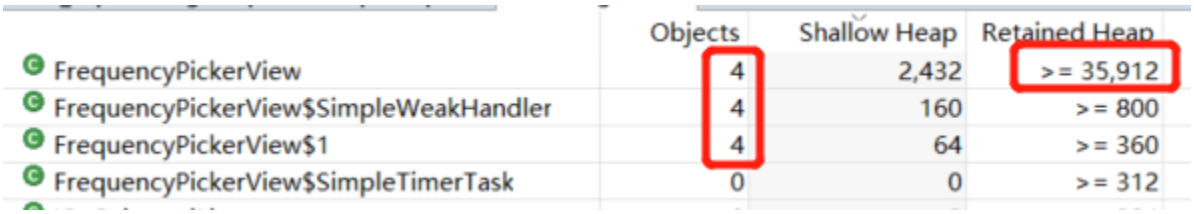
FrequencyPickerView创建了四个对象，因此看看该对象被哪些对象持有。



从Incoming图中可以看到SimpleWeakHandler持有FrequencyPickerView对象。下面2.1.5小节结合代码分析泄露原因和修复方法。

2.1.5 部分泄露点举例

1.Handler泄露解决方案



	Objects	Shallow Heap	Retained Heap
FrequencyPickerView	4	2,432	>= 35,912
FrequencyPickerView$SimpleWeakHandler	4	160	>= 800
FrequencyPickerView$1	4	64	>= 360
FrequencyPickerView$SimpleTimerTask	0	0	>= 312

通过上图可以看到深堆量比较大，因此怀疑内存泄露，查看对应代码，发现问题。

```
private class SimpleWeakHandler extends WeakHandler<FrequencyPickerView> {

    public SimpleWeakHandler(FrequencyPickerView owner) {
        super(owner);
    }

    @Override
    public void handleMessage(@NotNull Message msg) {
        super.handleMessage(msg);
        FrequencyPickerView view = getOwner();
        if (view.mScrollX > mSpeed) {
            if (mSelectListener != null) {    // 此处引用外部类对象，持有外部view对象!
                mSelectListener.onIsTuning(true);
            }
            move2NextSpeed(view);
```

```
        } else {
            ...
            if (offsetAngle % ITEM_ANGLE == 0) {
                if (view.mTask != null) {
                    view.mTask.cancel();
                    view.mTask = null;
                    if (mPressedChangeFre) {    // 此处引用外部类对象，持有外部view对
象!
                        view.performSelect();
                        mPressedChangeFre = false;
                    }
                }
                if (mSelectListener != null) {   // 此处引用外部类对象，持有外部view对
象!
                    mSelectListener.onIsTuning(false);
                }
            } else {
                if (Math.abs(offsetAngle) % ITEM_ANGLE >= ITEM_ANGLE / NUMBER_2)
{
                    move2NextFreqPoint(view, offsetAngle);
                } else if (Math.abs(view.mMoveTotalX) % view.mItemWidth <
view.mItemWidth / NUMBER_2) {
                    move2PrevFreqPoint(view, offsetAngle);
                }
            }
        }
        view.invalidate();
    }
}
```

分析：

从代码可以看到，其实该小伙伴是有考虑Handler内存泄露问题的，自己封装了一个WeakHandler，但是使用方法有一点问题。

```
// 自定义的WeakHandler，此类自定义没问题
public abstract class WeakHandler<T> extends Handler {
    private WeakReference<T> mOwner;

    public WeakHandler(T owner) {
        mOwner = new WeakReference<T>(owner);
    }

    public T getOwner() {
        return mOwner.get();
    }
}
```

问题出在上面的SimpleWeakHandler，在该handler中直接引用了外部view的成员变量。

代码修复建议：

- 使用static修饰class，防止内部类直接引用外部类成员变量
- 使用WeakReference包裹对象，当被回收之后，就不操作该对象了。

```
private static class SimpleWeakHandler extends WeakHandler<FrequencyPickerView>
{

    public SimpleWeakHandler(FrequencyPickerView owner) {
        super(owner);
    }

    @Override
    public void handleMessage(@NotNull Message msg) {
        super.handleMessage(msg);
        FrequencyPickerView view = getOwner();
        if(view == null) {
            return;    // 如果被回收了就啥也不干
        }
        // 下面需要操作外部view对象的成员变量，都通过view对象来引用，例如：view.xxx = xxx;
        //========================================
        view.xxx = xxx;   //运算view
        view.invalidate();
        //========================================
    }
}
```

PS：以上代码只是一个案例，请FO遍历工程代码，排查Handler内存泄露问题。另外解决内存泄露的handler写法可以参考模板代码4.3。

2.静态类持有Activity对象

此问题和上面view泄露没有本质区别，但是还是此小节分析一下Activity泄露。因此我看应用整个工程的架构是MVVM，采用MVVM可能存在的泄露点就是当界面被销毁之后，没有注销监听者。下面我们来看一下。

```
//
mediax\localradio\src\main\java\com\iflytek\autofly\localradio\ui\MainActivity.j
ava
public class MainActivity extends BaseActivity {

    private static final String TAG = "LR_MainActivity";
    private MainViewModel mMainViewModel;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        ActivityUtils.getInstance().addActivity(this);
        LRSingleLiveData.getInstance().observe(this, messageEvent ->
processExtraData()); //注册数据监听
    }


     @Override
    protected void onDestroy() {
        super.onDestroy();
        ActivityUtils.getInstance().removeActivity(this);
    }
}
```

从代码中看到该Activity在onCreate中注册了数据监听者，但是在onDestroy()中没有注销监听者。

```
     * LiveData keeps a strong reference to the observer and the owner as long
as the
     * given LifecycleOwner is not destroyed. When it is destroyed, LiveData
removes references to
     * the observer &amp;amp; the owner.
    // LiveData持有observer和owner强引用，需要在Activity销毁之后，注销监听者
    @MainThread
    public void observe(@NonNull LifecycleOwner owner, @NonNull Observer<? super
T> observer) {
    ......
    }
```

优化建议

```
// LRSingleLiveData.java重写以下这两个方法，在onDestroy()中释放observer和owner。
public abstract class LiveData<T> {
    @MainThread
    public void removeObserver(@NonNull final Observer<? super T> observer) {
        assertMainThread("removeObserver");
        ObserverWrapper removed = mObservers.remove(observer);
        if (removed == null) {
            return;
        }
        removed.detachObserver();
        removed.activeStateChanged(false);
    }

    @SuppressWarnings("WeakerAccess")
    @MainThread
    public void removeObservers(@NonNull final LifecycleOwner owner) {
        assertMainThread("removeObservers");
        for (Map.Entry<Observer<? super T>, ObserverWrapper> entry : mObservers)
{
            if (entry.getValue().isAttachedTo(owner)) {
                removeObserver(entry.getKey());
            }
        }
    }
}
```

2.2 Android内存泄漏常见场景以及解决方案

2.2.1、资源性对象未关闭

对于资源性对象不再使用时，应该立即调用它的close()函数，将其关闭，然后再置为null。例如Bitmap等资源未关闭会造成内存泄漏，此时我们应该在Activity销毁时及时关闭。

2.2.2、注册对象未注销

例如BraodcastReceiver、EventBus未注销造成的内存泄漏，我们应该在Activity销毁时及时注销。

2.2.3、类的静态变量持有大数据对象

尽量避免使用静态变量存储数据，特别是大数据对象，建议使用数据库存储。

2.2.4、单例造成的内存泄漏

优先使用Application的Context，如需使用Activity的Context，可以在传入Context时使用弱引用进行封装，然后，在使用到的地方从弱引用中获取Context，如果获取不到，则直接return即可。

2.2.5、非静态内部类的静态实例

该实例的生命周期和应用一样长，这就导致该静态实例一直持有该Activity的引用，Activity的内存资源不能正常回收。此时，我们可以将该内部类设为静态内部类或将该内部类抽取出来封装成一个单例，如果需要使用Context，尽量使用Application Context，如果需要使用Activity Context，就记得用完后置空让GC可以回收，否则还是会内存泄漏。

2.2.6、Handler临时性内存泄漏

Message发出之后存储在MessageQueue中，在Message中存在一个target，它是Handler的一个引用，Message在Queue中存在的时间过长，就会导致Handler无法被回收。如果Handler是非静态的，则会导致Activity或者Service不会被回收。并且消息队列是在一个Looper线程中不断地轮询处理消息，当这个Activity退出时，消息队列中还有未处理的消息或者正在处理的消息，并且消息队列中的Message持有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。解决方案如下所示：

- 1、使用一个静态Handler内部类，然后对Handler持有的对象（一般是Activity）使用弱引用，这样在回收时，也可以回收Handler持有的对象。
- 2、在Activity的Destroy或者Stop时，应该移除消息队列中的消息，避免Looper线程的消息队列中有待处理的消息需要处理。

需要注意的是，AsyncTask内部也是Handler机制，同样存在内存泄漏风险，但其一般是临时性的。对于类似AsyncTask或是线程造成的内存泄漏，我们也可以将AsyncTask和Runnable类独立出来或者使用静态内部类。

2.2.7、容器中的对象没清理造成的内存泄漏

在退出程序之前，将集合里的东西clear，然后置为null，再退出程序

2.2.8、WebView

WebView都存在内存泄漏的问题，在应用中只要使用一次WebView，内存就不会被释放掉。我们可以为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。

2.2.9、使用ListView时造成的内存泄漏

在构造Adapter时，使用缓存的convertView。

3.参考文献

1.lykhonis/ObjectPool: Object pool for Android (github.com)

2.添加bitmap相关的操作和工具

3.使用内存性能分析器查看应用的内存使用情况 | Android 开发者 | Android Developers (google.cn)

4.JVM 内存分析工具 MAT 的深度讲解与实践——入门篇

4.参考代码

4.1 对象池代码

```
// 第一步：在需要使用对象池的地方创建一个对象池，TestBean为对象池内存放的对象
// 默认对象池容量为4个，不建议修改为太大的值。
private ObjectPool mPool = new ObjectPool() {
        @Override
        protected Object create(Class<?> type) {
            return new TestBean();
        }
    };

// 第二步：从对象池获取一个对象，并且使用
TestBean tmp = mPool.acquire();
tmp.setAge(18);
tmp.setName("JackOu");

// 第三步：释放临时对象
mPool.release(tmp);
```

```
// ObjectPool.java 对象池类代码

import androidx.collection.SimpleArrayMap;

/**
 * Object Pool is thread-safe pattern to simplify access and reuse common
objects. Particular object
 * pool supports creation of object by using factory pattern as well as multiple
type of object sets
 */
public class ObjectPool {

    static final int POOL_INITIAL_CAPACITY = 4;

    static final class DefaultClass {}

    static final Class<?> DEFAULT_TYPE = DefaultClass.class;

    final SimpleArrayMap<Class<?>, Object[]> mPool;
```

```
    Object[] mInuse;
    Factory mFactory;

    /**
     * Create empty thread-safe object pool.
     */
    public ObjectPool() {
        this(null);
    }

    /**
     * Create empty thread-safe object pool
     *
     * @param factory Factory
     */
    public ObjectPool(Factory factory) {
        mFactory = factory;
        mPool = new SimpleArrayMap<>(POOL_INITIAL_CAPACITY);
        mInuse = new Object[POOL_INITIAL_CAPACITY];
    }

    /**
     * Acquire object in pool or create new if does not exist
     *
     * @param type Type of object set
     * @return Object from set type
     */
    @SuppressWarnings("unchecked")
    public <T> T acquire(Class<T> type) {
        synchronized (mPool) {
            Object[] pool = mPool.get(type);
            if (pool == null) {
                mPool.put(type, pool = new Object[POOL_INITIAL_CAPACITY]);
            }
            Object object = null;
            int size = pool.length;
            for (int i = 0; i < size; i++) {
                if (pool[i] != null) {
                    object = pool[i];
                    pool[i] = null;
                    break;
                }
            }
            if (object == null && (object = create(type)) == null) {
                throw new NullPointerException("Create has to return non-null
object!");
            }
            size = mInuse.length;
            for (int i = 0; i < size; i++) {
                if (mInuse[i] == null) {
                    return (T) (mInuse[i] = object);
                }
            }
            mInuse = grow(mInuse, idealObjectArraySize(size * 2));
            return (T) (mInuse[size] = object);
        }
    }
```

```
    int inuse() {
        int size = 0;
        for (Object object : mInuse) {
            if (object != null) size++;
        }
        return size;
    }

    int sizeDefault() {
        return size(DEFAULT_TYPE);
    }

    int size(Class<?> type) {
        int size = 0;
        Object[] pool = mPool.get(type);
        if (pool != null) {
            for (Object object : pool) {
                if (object != null) size++;
            }
        }
        return size;
    }

    /**
     * Removes all objects of set type
     *
     * @param type Type of object set
     */
    public void clear(Class<?> type) {
        synchronized (mPool) {
            Object[] pool = mPool.get(type);
            if (pool != null) clear(pool);
        }
    }

    /**
     * Removes all objects and sets
     */
    public void clear() {
        synchronized (mPool) {
            int size = mPool.size();
            for (int i = 0; i < size; i++) {
                Object[] pool = mPool.valueAt(i);
                if (pool != null) clear(pool);
            }
        }
    }

    /**
     * Acquire object in pool or create new if does not exist
     *
     * @return Object from set type
     */
    @SuppressWarnings("unchecked")
    public <T> T acquire() {
        return (T) acquire(DEFAULT_TYPE);
    }
```

```
    /**
     * Release object acquired from pool back
     *
     * @param object Object to release back to pool
     */
    public void release(Object object) {
        synchronized (mPool) {
            int index = indexOf(mInuse, object);
            if (object != null && index >= 0) {
                mInuse[index] = null;
                Class<?> type = object.getClass();
                if (!mPool.containsKey(type)) type = DEFAULT_TYPE;
                Object[] pool = mPool.get(type);
                int size = pool.length;
                for (int i = 0; i < size; i++) {
                    if (pool[i] == null) {
                        pool[i] = object;
                        return;
                    }
                }
                pool = grow(pool, idealObjectArraySize(size * 2));
                pool[size] = object;
                mPool.put(type, pool);
            }
        }
    }

    /**
     * Create new object for type set
     *
     * @param type Type of object set
     * @return Non-null object
     */
    protected Object create(Class<?> type) {
        return mFactory == null ? null : mFactory.create(type);
    }

    /**
     * Factory to create objects for pool
     */
    public interface Factory {
        /**
         * Create new object for type set
         *
         * @param type Type of object set
         * @return Non-null object
         */
        Object create(Class<?> type);
    }

    static int indexOf(Object[] array, Object object) {
        int size = array.length;
        for (int i = 0; i < size; i++) {
            if (array[i] == object) return i;
        }
        return -1;
    }
```

```
    static void clear(Object[] array) {
        int size = array.length;
        for (int i = 0; i < size; i++) {
            array[i] = null;
        }
    }

    static Object[] grow(Object[] array, int size) {
        Object[] result = new Object[size];
        System.arraycopy(array, 0, result, 0, array.length);
        return result;
    }

    static int idealObjectArraySize(int need) {
        return idealByteArraySize(need * 4) / 4;
    }

    static int idealByteArraySize(int need) {
        for (int i = 4; i < 32; i++)
            if (need <= (1 << i) - 12)
                return (1 << i) - 12;
        return need;
    }
}
```

4.2 自定义Bitmap加载类

关于bitmap相关的一些操作，详见《参考文献3.2》,仅供相互交流参考。

```
public class BitmapLoadTool {

    //======================从本地（SDcard）文件读取=======================
    /**
     * 获取缩放后的本地图片
     *
     * @param filePath 文件路径
     * @param width    宽
     * @param height   高
     * @return
     */
    public static Bitmap readBitmapFromFile(String filePath, int width, int
height) {
        BitmapFactory.Options options = new BitmapFactory.Options();
        options.inJustDecodeBounds = true;
        BitmapFactory.decodeFile(filePath, options);
        float srcWidth = options.outWidth;
        float srcHeight = options.outHeight;
        int inSampleSize = 1;
        if (srcHeight > height || srcWidth > width) {
            if (srcWidth > srcHeight) {
                inSampleSize = Math.round(srcHeight / height);
            } else {
                inSampleSize = Math.round(srcWidth / width);
            }
        }
```

```
        options.inJustDecodeBounds = false;
        options.inSampleSize = inSampleSize;
        return BitmapFactory.decodeFile(filePath, options);
    }

    /**
     * 获取缩放后的本地图片
     * readBitmapFromFileDescriptor()效率高于readBitmapFromFile()
     *
     * @param filePath 文件路径
     * @param width    宽
     * @param height   高
     * @return
     */
    public static Bitmap readBitmapFromFileDescriptor(String filePath, int
width, int height) {
        try {
            FileInputStream fis = new FileInputStream(filePath);
            BitmapFactory.Options options = new BitmapFactory.Options();
            options.inJustDecodeBounds = true;
            BitmapFactory.decodeFileDescriptor(fis.getFD(), null, options);
            float srcWidth = options.outWidth;
            float srcHeight = options.outHeight;
            int inSampleSize = 1;
            if (srcHeight > height || srcWidth > width) {
                if (srcWidth > srcHeight) {
                    inSampleSize = Math.round(srcHeight / height);
                } else {
                    inSampleSize = Math.round(srcWidth / width);
                }
            }
            options.inJustDecodeBounds = false;
            options.inSampleSize = inSampleSize;
            return BitmapFactory.decodeFileDescriptor(fis.getFD(), null,
                    options);
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        return null;
    }

    //======================从输入流中读取文件（网络加载）=======================
    /**
     * 获取缩放后的本地图片
     *
     * @param ins    输入流
     * @param width  宽
     * @param height 高
     * @return
     */
    public static Bitmap readBitmapFromInputStream(InputStream ins, int width,
int height) {
        BitmapFactory.Options options = new BitmapFactory.Options();
        options.inJustDecodeBounds = true;
        BitmapFactory.decodeStream(ins, null, options);
        float srcWidth = options.outWidth;
        float srcHeight = options.outHeight;
        int inSampleSize = 1;
```

```
        if (srcHeight > height || srcWidth > width) {
            if (srcWidth > srcHeight) {
                inSampleSize = Math.round(srcHeight / height);
            } else {
                inSampleSize = Math.round(srcWidth / width);
            }
        }
        options.inJustDecodeBounds = false;
        options.inSampleSize = inSampleSize;
        return BitmapFactory.decodeStream(ins, null, options);
    }

    //=====================Resource资源加载=====================
    /**
     * 使用decodeResource()相对比较消耗内存，建议用decodeStream()
     *
     * @param resources
     * @param resourcesId
     * @param width
     * @param height
     * @return
     */
    public static Bitmap readBitmapFromResourceUsingDecodeResource(Resources
resources, int resourcesId, int width, int height) {
        BitmapFactory.Options options = new BitmapFactory.Options();
        options.inJustDecodeBounds = true;
        BitmapFactory.decodeResource(resources, resourcesId, options);
        float srcWidth = options.outWidth;
        float srcHeight = options.outHeight;
        int inSampleSize = 1;
        if (srcHeight > height || srcWidth > width) {
            if (srcWidth > srcHeight) {
                inSampleSize = Math.round(srcHeight / height);
            } else {
                inSampleSize = Math.round(srcWidth / width);
            }
        }
        options.inJustDecodeBounds = false;
        options.inSampleSize = inSampleSize;
        return BitmapFactory.decodeResource(resources, resourcesId, options);
    }

    public static Bitmap readBitmapFromResourceUsingDecodeStream(Resources
resources, int
            resourcesId, int width, int height) {
        InputStream ins = resources.openRawResource(resourcesId);
        BitmapFactory.Options options = new BitmapFactory.Options();
        options.inJustDecodeBounds = true;
        BitmapFactory.decodeStream(ins, null, options);
        float srcWidth = options.outWidth;
        float srcHeight = options.outHeight;
        int inSampleSize = 1;
        if (srcHeight > height || srcWidth > width) {
            if (srcWidth > srcHeight) {
                inSampleSize = Math.round(srcHeight / height);
            } else {
                inSampleSize = Math.round(srcWidth / width);
            }
```

```
        }
        options.inJustDecodeBounds = false;
        options.inSampleSize = inSampleSize;
        return BitmapFactory.decodeStream(ins, null, options);
    }

    //======================Assets资源加载方式=======================
    /**
     * 获取缩放后的本地图片
     *
     * @param filePath 文件路径,即文件名称
     * @return
     */
    public static Bitmap readBitmapFromAssetsFile(Context context, String
            filePath) {
        Bitmap image = null;
        AssetManager am = context.getResources().getAssets();
        try {
            InputStream is = am.open(filePath);
            image = BitmapFactory.decodeStream(is);
            is.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        return image;
    }

    //======================从二进制数据读取图片=======================
    public static Bitmap readBitmapFromByteArray(byte[] data, int width, int
height) {
        BitmapFactory.Options options = new BitmapFactory.Options();
        options.inJustDecodeBounds = true;
        BitmapFactory.decodeByteArray(data, 0, data.length, options);
        float srcWidth = options.outWidth;
        float srcHeight = options.outHeight;
        int inSampleSize = 1;
        if (srcHeight > height || srcWidth > width) {
            if (srcWidth > srcHeight) {
                inSampleSize = Math.round(srcHeight / height);
            } else {
                inSampleSize = Math.round(srcWidth / width);
            }
        }
        options.inJustDecodeBounds = false;
        options.inSampleSize = inSampleSize;
        return BitmapFactory.decodeByteArray(data, 0, data.length, options);
    }
}
```

4.3 Handler解决内存泄露的模板写法

```
public class MainActivityLeak extends AppCompatActivity {

    TextView mTextView;
    MyHandler mMyHandler;
```

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main_leak);
        mTextView = findViewById(R.id.tv_handler);
        mMyHandler = new MyHandler(MainActivityLeak.this);
    }

    // 使用static修饰类，防止内部类直接引用外部类成员变量
    // WeakReference修饰外部类对象，如果被回收了，就取消操作
    private static class MyHandler extends Handler {
        private WeakReference<MainActivityLeak> mWeakReference;

        public MyHandler(MainActivityLeak activity) {
            mWeakReference = new WeakReference<>(activity);
        }

        @Override
        public void handleMessage(Message msg) {
            super.handleMessage(msg);
            MainActivityLeak mainActivity = mWeakReference.get();
            switch (msg.what) {
                case 1:
                    if (mainActivity != null) {
                        mainActivity.mTextView.setText(msg.obj + "");
                    }
                    break;
                default:
                    break;
            }
        }
    }
}
```