前言

Android的严重碎片化，通常会给开发人员造成很大的苦恼！经过测试验证后的版本，一旦发布出去，会收到很多奇葩的反馈，在Bugly崩溃分析平台上也总会出现很多意想不到的问题。

有的可以很容易从堆栈信息中定位到出现问题的代码，比较容易解决。但是也会经常出现一些只有系统代码堆栈的问题，这类问题定位往往都比较困难。对于这些疑难杂症，有些问题解决的方法是比较通用的。这里我整理了一些跟进的思路，抛砖引玉。

以手机管家某个版本在Bugly崩溃分析平台上发现的一个上报量比较大的问题为例，出错的堆栈信息如下所示：

```
java.lang.IllegalArgumentException: Window type can not be changed after the window is added.
android.os.Parcel.readException(Parcel.java:1429)
android.os.Parcel.readException(Parcel.java:1379)
android.view.IWindowSession$Stub$Proxy.relayout(IWindowSession.java:813)
android.view.ViewRootImpl.relayoutWindow(ViewRootImpl.java:4472)
android.view.ViewRootImpl.performTraversals(ViewRootImpl.java:1642)
android.view.ViewRootImpl.doTraversal(ViewRootImpl.java:1204)
android.view.ViewRootImpl$TraversalRunnable.run(ViewRootImpl.java:4951)
android.view.Choreographer$CallbackRecord.run(Choreographer.java:776)
android.view.Choreographer.doCallbacks(Choreographer.java:579)
android.view.Choreographer.doFrame(Choreographer.java:548)
android.view.Choreographer$FrameDisplayEventReceiver.run(Choreographer.java:762)
android.os.Handler.handleCallback(Handler.java:725)
android.os.Handler.dispatchMessage(Handler.java:92)
android.os.Looper.loop(Looper.java:153)
android.app.ActivityThread.main(ActivityThread.java:5383)
java.lang.reflect.Method.invokeNative(Native Method)
java.lang.reflect.Method.invoke(Method.java:511)
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:853)
com.android.internal.os.ZygoteInit.main(ZygoteInit.java:620)
dalvik.system.NativeStart.main(Native Method)
```

初看崩溃的堆栈信息，大概只能看到是View绘制过程出现错误，没有明确信息可以找到出错的代码位置，类似这样的问题怎样来定位呢？

1 通过Android源码分析问题根源

先从系统源码找到出现这个异常的地方，看看能不能找到什么线索，从代码堆栈中看到在`ViewRootImpl`类出现问题，先找到这个类的`relayoutWindow`方法，代码截图如下（备注：Android4.4源码）：

```
private int relayoutWindow(WindowManager.LayoutParams params, int viewVisibility,
        boolean insetsPending) throws RemoteException {

    float appScale = mAttachInfo.mApplicationScale;
    boolean restore = false;
    if (params != null && mTranslator != null) {
        restore = true;
        params.backup();
        mTranslator.translateWindowLayout(params);
    }
    if (params != null) {
        if (DBG) Log.d(TAG, "WindowLayout in layoutWindow:" + params);
    }
    mPendingConfiguration.seq = 0;
    //Log.d(TAG, ">>>>>> CALLING relayout");
    if (params != null && mOrigWindowType != params.type) {
        // For compatibility with old apps, don't crash here.
        if (mTargetSdkVersion < android.os.Build.VERSION_CODES.ICE_CREAM_SANDWICH) {
            Slog.w(TAG, "Window type can not be changed after "
                    + "the window is added; ignoring change of " + mView);
            params.type = mOrigWindowType;
        }
    }
    int relayoutResult = mWindowSession.relayout(
            mWindow, mSeq, params,
            (int) (mView.getMeasuredWidth() * appScale + 0.5f),
            (int) (mView.getMeasuredHeight() * appScale + 0.5f),
            viewVisibility, insetsPending ? WindowManagerGlobal.RELAYOUT_INSETS_PENDING : 0,
            mWinFrame, mPendingContentInsets, mPendingVisibleInsets,
            mPendingConfiguration, mSurface);
    //Log.d(TAG, "<<<<<< BACK FROM relayout");
    if (restore) {
        params.restore();
    }
```

上面代码第一个红色框出现了"`window type can not be changed after the window is added`"这句话，跟崩溃的信息类似。但仔细看一遍，不是这个地方抛出来的异常，这里只是一句Log输出，而且是对`ICE_CREAM_SANDWICH`以下的系统（也就是4.0以下系统）才会输出这样的Log。
这里先留个疑问，**为什么对系统版本号进行判断做不同的处理？**

往下看，这个方法最后是调用了`mWindowSession.relayout`方法，通过分析，`mWindowSession`是实现了`IWindowManager`接口，这是一个IPC调用，接下来找到`WindowManagerService`类的`relayoutWindow`方法，如下图：

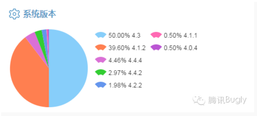
从这里终于看到了抛出异常的地方，与崩溃堆栈相吻合，正是用IPC调用`WindowManager`的方法时出现的异常，通过对源码的阅读，问题的根源也很明显了，window显示出来后`LayoutType`是不能再修改的，否则在绘制时判断到type被修改就会抛出以上异常。了解了问题根源后，定位问题有明确的方向，在项目源码中搜索修改`LayoutParam.type`的代码，很快可以分析出造成崩溃的地方。

2 检查Android系统版本兼容性

继续来看上述的崩溃问题，在前面分析源码过程中，留了一个问题，**为什么对系统版本号进行判断做不同的处理？**
源码中对于`ICE_CREAM_SANDWICH`以下系统有特殊的处理逻辑，再找到Android2.3的`WindowManagerService`源码来看，2.3系统的处理方式果然是不同的，不会有这个异常的抛出。也就是说，同样的代码，在4.0以下的系统运行是正常的。由此可以看到，分析崩溃问题时也需要从系统兼容这个角度来分析，可以关注下上报的数据是分布在不同的系统版本中还是明显集中在某些版本上出现问题，有助于快速界定问题的影响范围以及找到对应解决方法。

在项目中，有时候会使用系统隐藏API实现一些功能，由于这些是非公开的，在Android版本更新过程中，很有可能被修改到，容易出现版本不兼容问题，所以平时要尽量避免使用。另外使用高版本的SDK开发完成后，高版本中可能会有新增的API是低版本中不存在的，这时候会出现比较常见的`java.lang.VerifyError`类型的异常，如果想兼容性更好，也要多测试下在低版本中有没有问题。

在这里插一句，在Bugly崩溃分析平台上，对发生的崩溃问题是有系统、机型等兼容分析后的数据，这有助于开发在分析问题时，可以快速确定是否与系统或机型相关，提高解决问题效率。大家在选择崩溃分析工具时，也可以关注下这点。



3 确认是否机型或ROM适配问题

由于Android是开源的系统，厂商获拿到源码后可以自由地定制和改造，当然，修改的过程可能带来一些不稳定的问题，会导致同样的代码，在特定的机型出现适配的问题。

来看一个很普遍的崩溃，异常信息如下所示：

```
java.lang.NoSuchMethodError: android.app.ANRAppManager.dumpMessageHistory
android.app.ActivityThread$ApplicationThread.dumpMessageHistory(ActivityThread.java:1177)
android.app.ApplicationThreadNative.onTransact(ApplicationThreadNative.java:609)
android.os.Binder.execTransact(Binder.java:351)
dalvik.system.NativeStart.run(Native Method)
```

先按照上面提到的思路，通过Android源码来分析问题的根源，找到`ActivityThread`类，仔细检查，看到这个类里面并没有`dumpMessageHistory`这个方法，也没有`ANRAppManager`类，很明显这不是Android原生系统的方法，是被修改过的。

再检查下出现这个崩溃的机型特点，发现也不是集中在某些机型上出现，但是仔细再分析下，就能看到这些机型基本都是使用MTK芯片的，因为看到ROM的`fingerprint`，发现出现这个崩溃的基本是"**alps/**"开头的，从ROM的指纹可以猜测大概是MTK修改的一个Android版本。

对于这个问题，我们找到了一台出现这个崩溃的机器并重现出来了，得到的崩溃堆栈信息跟上面完全一样。实际上这个问题是在程序中制造了一个ANR，系统有Bug导致在执行dump信息的方法时出现崩溃。要解决这个问题，就只能去分析程序为什么会出现ANR了，这时候可以获取到main线程的执行状态，来判断程序是不是出现阻塞，导致响应超时，至于ANR问题的分析，是另外一个话题了，这里先不展开。

4 通过辅助信息缩小排查范围

在代码量比较大的项目，排查问题相对会比较困难，那么在定位一些崩溃问题时候，要尽量获取更多的辅助信息，把排查范围尽量地缩小，来看下面这个问题：

```
java.lang.RuntimeException: Canvas: trying to use a recycled bitmap android.graphics.Bitmap@41dec440
android.graphics.Canvas.throwIfRecycled(Canvas.java:1026)
android.graphics.Canvas.drawBitmap(Canvas.java:1127)
android.graphics.drawable.BitmapDrawable.draw(BitmapDrawable.java:393)
android.widget.ImageView.onDraw(ImageView.java:967)
android.view.View.draw(View.java:13818)
android.view.View.draw(View.java:13702)
android.view.ViewGroup.drawChild(ViewGroup.java:3025)
android.view.ViewGroup.dispatchDraw(ViewGroup.java:2889)
android.view.View.draw(View.java:13700)
android.view.ViewGroup.drawChild(ViewGroup.java:3025)
android.view.ViewGroup.dispatchDraw(ViewGroup.java:2889)
android.widget.AbsListView.dispatchDraw(AbsListView.java:2449)
android.view.View.draw(View.java:13821)
android.widget.AbsListView.draw(AbsListView.java:4124)
android.view.View.draw(View.java:13702)
android.view.ViewGroup.drawChild(ViewGroup.java:3025)
android.view.ViewGroup.dispatchDraw(ViewGroup.java:2889)
android.view.View.draw(View.java:13700)
android.view.ViewGroup.drawChild(ViewGroup.java:3025)
android.view.ViewGroup.dispatchDraw(ViewGroup.java:2889)
```

这个问题是在UI绘制过程中，由于要绘制的图片已经被回收而出现的崩溃，可能是由于程序中对图片内存的回收时机不对。一般情况下是因为避免程序中使用的图片占用太多的内存，采取了主动回收的策略，但是对界面的生命周期和图片回收的时机理解有误造成的。如果多个界面都用到了图片回收的策略，那么就很难找到是哪个地方出问题了。

因为上面的崩溃堆栈信息只有系统的代码，没有跟应用层关联的代码，这时候就要考虑如何把定位问题的范围缩小。**采取的解决方法**是，程序运行过程中会记录当前显示的是哪一个界面，当出现崩溃的时候就把最后显示的界面信息也一并上报上来，那么定位问题就比较容易了，基本是确定了是在哪个界面出现的，剩下的就是对这个界面相关的代码进行排查，而不用整个项目的代码都来排查。

记录定位问题的辅助信息对于跟进疑难问题很有帮助，相当于程序运行的关键日志，可以了解出现崩溃时程序更多的状态，而不只是一个崩溃的代码堆栈，这个过程要靠经验积累，把一些认为比较重要的代码路径形成定位问题的日志，辅助分析能更快地找到问题根源。

5 根据出现崩溃的线程名排查问题

上面讲到的通过辅助信息来定位问题，而通过崩溃的线程名称也能够把问题出现的范围缩小。Android的UI绘制是在main线程了，main线程如果被阻塞太长时间会出现ANR问题，所以利用工作线程来处理耗时任务是用得很多的。

如果在编程的过程中形成好的习惯，在线程任务运行时，给当前线程设置一个特定的名称，那么在出现崩溃的时候就很容易根据线程的名称看到是在执行哪个任务出现的问题，这个方法也是适用于使用线程池的项目，可以在线程开始执行时赋予一个当前执行的任务名称，出现问题后就可以根据线程名称来找到对应的代码段。

6 其它的方法

除了上述讲的几种情况，还有一些问题，并不是能够很明确地分析到产生的原因，比如内存溢出的问题（**java.lang.OutOfMemoryError**）。内存的问题在不同的机型上会有不同的表现，低端机型容易出现内存不足而崩溃，这就要求开发人员要想办法对内存占用进行优化，特别是在大内存的申请，比如图片显示处理等等，要控制好内存的峰值，防止超出系统限制而出现OOM问题。

项目过程中，类似这样的问题也很多，需要多挖掘软硬件环境的数据，找到更多的线索和规律，既然出现了问题，说明有哪个环节处理得不当或者需要优化，Android系统的稳定性优化的道路任重而道远。