本文将会对 Android 内存优化相关的知识进行总结以及最后案例分析(一二部分是理论知识总结，你也可以直接跳到第三部分看案例):

一、 Android 内存分配回收机制
二、Android 常见内存问题和对应检测，解决方式
三、 JOOX 内存优化案例
四、总结

工欲善其事必先利其器，想要优化 App 的内存占用，那么还是需要先了解 Android 系统的内存分配和回收机制。

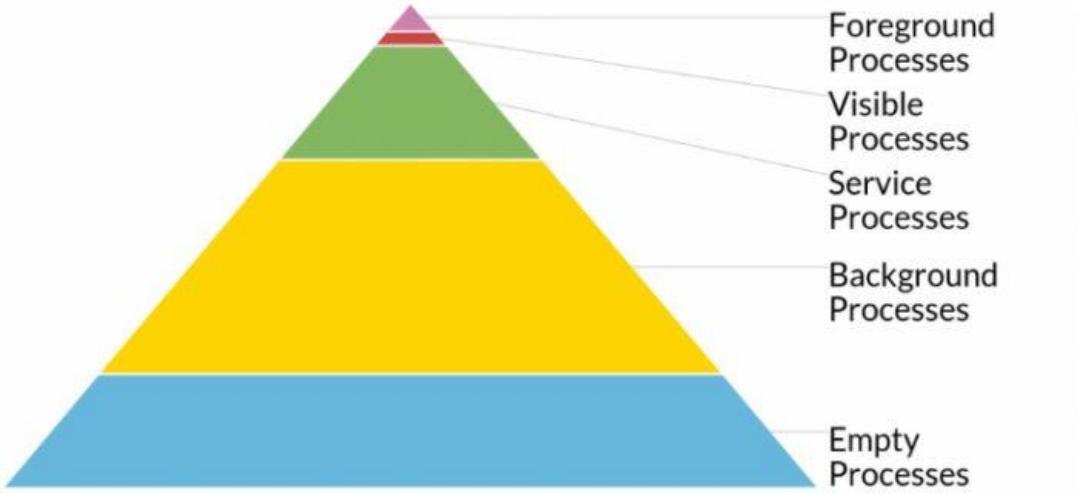
一 ，Android 内存分配回收机制

参考 Android 操作系统的内存回收机制[1]，这里简单做下总结:

从宏观角度上来看 Android 系统可以分为三个层次

Application Framework
Dalvik 虚拟机
Linux 内核
这三个层次都有各自内存相关工作:

1. Application Framework
Anroid 基于进程中运行的组件及其状态规定了默认的五个回收优先级:



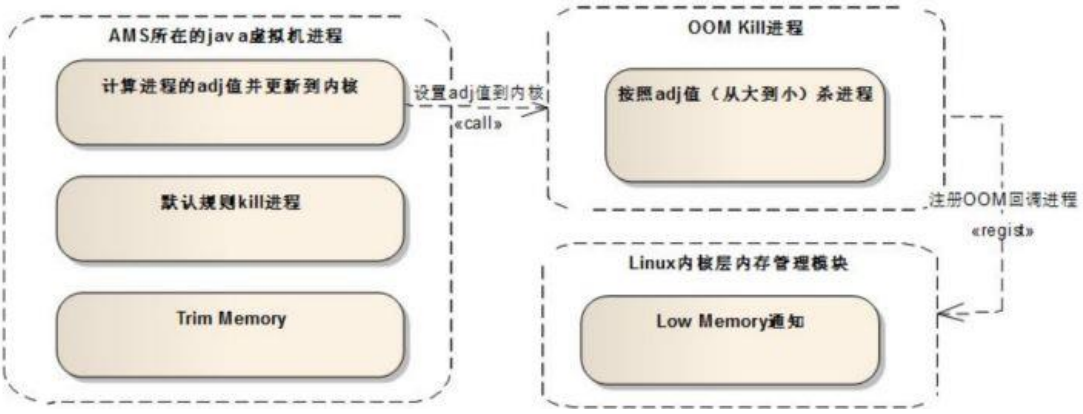
Empty process(空进程)
Background process(后台进程)
Service process(服务进程)
Visible process(可见进程)
Foreground process(前台进程)
系统需要进行内存回收时最先回收空进程,然后是后台进程，以此类推最后才会回收前台进

程（一般情况下前台进程就是与用户交互的进程了,如果连前台进程都需要回收那么此时系统几乎不可用了）。

由此也衍生了很多进程保活的方法（提高优先级，互相唤醒，native 保活等等），出现了国内各种全家桶，甚至各种杀不死的进程。

Android 中由 ActivityManagerService 集中管理所有进程的内存资源分配。

2. Linux 内核



参考 QCon 大会上阿里巴巴的 Android 内存优化分享[2]，这里最简单的理解就是 ActivityManagerService 会对所有进程进行评分（存放在变量 adj 中），然后再讲这个评分更新到内核，由内核去完成真正的内存回收(lowmemorykiller, Oom_killer)。这里只是大概的流程，中间过程还是很复杂的，有兴趣的同学可以一起研究，代码在系统源码 ActivityManagerService.java 中。

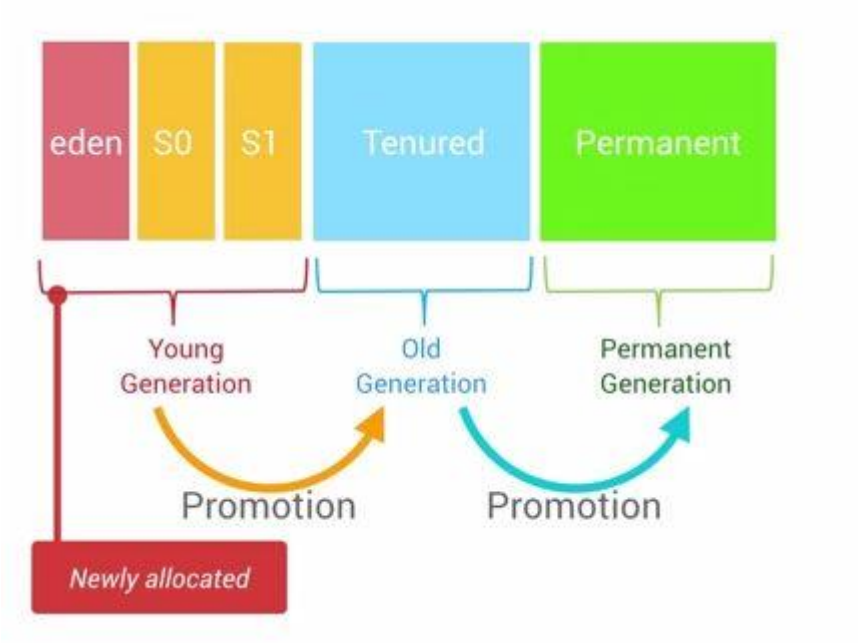
3. Dalvik 虚拟机

Android 进程的内存管理分析[3]，对 Android 中进程内存的管理做了分析。

Android 中有 Native Heap 和 Dalvik Heap。Android 的 Native Heap 言理论上可分配的空间取决了硬件 RAM，而对于每个进程的 Dalvik Heap 都是有大小限制的，具体策略可以看看 android dalvik heap 浅析[4]。

Android App 为什么会 OOM 呢？其实就是申请的内存超过了 Dalvik Heap 的最大值。这里也诞生了一些比较"黑科技"的内存优化方案，比如将耗内存的操作放到 Native 层，或者使用分进程的方式突破每个进程的 Dalvik Heap 内存限制。

Android Dalvik Heap 与原生 Java 一样，将堆的内存空间分为三个区域，Young Generation，Old Generation， Permanent Generation。



最近分配的对象会存放在 Young Generation 区域，当这个对象在这个区域停留的时间达到一定程度，它会被移动到 Old Generation，最后累积一定时间再移动到 Permanent Generation 区域。系统会根据内存中不同的内存数据类型分别执行不同的 gc 操作。

GC 发生的时候，所有的线程都是会被暂停的。执行 GC 所占用的时间和它发生在哪一个 Generation 也有关系，Young Generation 中的每次 GC 操作时间是最短的，Old Generation 其次，Permanent Generation 最长。

GC 时会导致线程暂停，导致卡顿，Google 在新版本的 Android 中优化了这个问题， 在 ART 中对 GC 过程做了优化揭秘 ART 细节 —— Garbage collection[5]，据说内存分配的效率提高了 10 倍，GC 的效率提高了 2-3 倍（可见原来效率有多低），不过主要还是优化中断和阻塞的时间，频繁的 GC 还是会导致卡顿。

上面就是 Android 系统内存分配和回收相关知识，回过头来看，现在各种手机厂商鼓吹人工智能手机，号称 18 个月不卡顿，越用越快，其实很大一部分 Android 系统的内存优化有关，无非就是利用一些比较成熟的基于统计，机器学习的算法定时清理数据，清理内存，甚至提前加载数据到内存。

二 ，Android 常见内存问题和对应检测，解决方式

1. 内存泄露

不止 Android 程序员，内存泄露应该是大部分程序员都遇到过的问题，可以说大部分的内存问题都是内存泄露导致的，Android 里也有一些很常见的内存泄露问题[6]，这里简单罗列下：

单例（主要原因还是因为一般情况下单例都是全局的，有时候会引用一些实际生命周期比较短的变量，导致其无法释放）
静态变量（同样也是因为生命周期比较长）
Handler 内存泄露[7]

匿名内部类（匿名内部类会引用外部类，导致无法释放，比如各种回调）
资源使用完未关闭（BraodcastReceiver，ContentObserver，File，Cursor，Stream，Bitmap）

对 Android 内存泄露业界已经有很多优秀的组件其中 LeakCanary 最为知名(Square 出品，Square 可谓 Android 开源界中的业界良心，开源的项目包括 okhttp, retrofit，otto, picasso, Android 开发大神 Jake Wharton 就在 Square)，其原理是监控每个 activity，在 activity ondestory 后，在后台线程检测引用，然后过一段时间进行 gc，gc 后如果引用还在，那么 dump 出内存堆栈，并解析进行可视化显示。使用 LeakCanary 可以快速地检测出 Android 中的内存泄露。

正常情况下，解决大部分内存泄露问题后，App 稳定性应该会有很大提升，但是有时候 App 本身就是有一些比较耗内存的功能，比如直播，视频播放，音乐播放，那么我们还有什么能做的可以降低内存使用，减少 OOM 呢？

2. 图片分辨率相关
分辨率适配问题。很多情况下图片所占的内存在整个 App 内存占用中会占大部分。我们知道可以通过将图片放到 hdpi/xhdpi/xxhdpi 等不同文件夹进行适配，通过 xml android:background 设置背景图片，或者通过 BitmapFactory.decodeResource()方法，图片实际上默认情况下是会进行缩放的。在 Java 层实际调用的函数都是或者通过 BitmapFactory 里的 decodeResourceStream 函数

```
public static Bitmap decodeResourceStream(Resources res, TypedValue value,
        InputStream is, Rect pad, Options opts) {

    if (opts == null) {
        opts = new Options();
    }

    if (opts.inDensity == 0 && value != null) {
        final int density = value.density;
        if (density == TypedValue.DENSITY_DEFAULT) {
            opts.inDensity = DisplayMetrics.DENSITY_DEFAULT;
        } else if (density != TypedValue.DENSITY_NONE) {
            opts.inDensity = density;
        }
    }

    if (opts.inTargetDensity == 0 && res != null) {
        opts.inTargetDensity = res.getDisplayMetrics().densityDpi;
    }

    return decodeStream(is, pad, opts);
 }
```

decodeResource 在解析时会对 Bitmap 根据当前设备屏幕像素密度 densityDpi 的值进行缩放适配操作，使得解析出来的 Bitmap 与当前设备的分辨率匹配，达到一个最佳的显示效果，

并且 Bitmap 的大小将比原始的大，可以参考下腾讯 Bugly 的详细分析 Android 开发绕不过的坑：你的 Bitmap 究竟占多大内存?

关于 Density、分辨率、-hdpi 等 res 目录之间的关系：

DensityDpi	分辨率	res	Density
160dpi	320×533	mdpi	1
240dpi	480×800	hdpi	1.5
320dpi	720×1280	xhdpi	2
480dpi	1080×1920	xxhdpi	3
560dpi	1440×2560	xxxhdpi	3.5

举个例子，对于一张 1280×720 的图片，如果放在 xhdpi，那么 xhdpi 的设备拿到的大小还是 1280×720 而 xxhpi 的设备拿到的可能是 1920×1080，这两种情况在内存里的大小分别为：3.68M 和 8.29M，相差 4.61M，在移动设备来说这几 M 的差距还是很大的。

尽管现在已经有比较先进的图片加载组件类似 Glide，Facebook Freso，或者老牌 Universal-Image-Loader，但是有时就是需要手动拿到一个 bitmap 或者 drawable，特别是在一些可能会频繁调用的场景(比如 ListView 的 getView)，怎样尽可能对 bitmap 进行复用呢？这里首先需要明确的是对同样的图片，要 尽可能复用，我们可以简单自己用 WeakReference 做一个 bitmap 缓存池，也可以用类似图片加载库写一个通用的 bitmap 缓存池，可以参考 GlideBitmapPool[8]的实现。

我们也来看看系统是怎么做的，对于类似在 xml 里面直接通过 android:background 或者 android:src 设置的背景图片，以 ImageView 为例，最终会调用 Resource.java 里的 loadDrawable:

```
 Drawable loadDrawable(TypedValue value, int id, Theme theme) throws NotFoundException
{

        // Next, check preloaded drawables. These may contain unresolved theme
        // attributes.
        final ConstantState cs;
        if (isColorDrawable) {
            cs = sPreloadedColorDrawables.get(key);
        } else {
            cs = sPreloadedDrawables[mConfiguration.getLayoutDirection()].get(key);
        }

        Drawable dr;
        if (cs != null) {
            dr = cs.newDrawable(this);
        } else if (isColorDrawable) {
```

```
            dr = new ColorDrawable(value.data);
        } else {
            dr = loadDrawableForCookie(value, id, null);
        }

        ...

        return dr;
    }
```

可以看到实际上系统也是有一份全局的缓存, sPreloadedDrawables, 对于不同的 drawable, 如果图片时一样的，那么最终只会有一份 bitmap(享元模式)，存放于 BitmapState 中，获取 drawable 时，系统会从缓存中取出这个 bitmap 然后构造 drawable。而通过 BitmapFactory.decodeResource()则每次都会重新解码返回 bitmap。所以其实我们可以通过 context.getResources().getDrawable 再从 drawable 里获取 bitmap，从而复用 bitmap，然而这里也有一些坑，比如我们获取到的这份 bitmap，假如我们执行了 recycle 之类的操作，但是假如在其他地方再使用它是那么就会有"Canvas: trying to use a recycled bitmap android.graphics.Bitmap"异常。

3. 图片压缩

BitmapFactory 在解码图片时，可以带一个 Options，有一些比较有用的功能，比如：

inTargetDensity 表示要被画出来时的目标像素密度

inSampleSize 这个值是一个 int，当它小于 1 的时候，将会被当做 1 处理，如果大于 1，那么就会按照比例（1 / inSampleSize）缩小 bitmap 的宽和高、降低分辨率，大于 1 时这个值将会被处置为 2 的倍数。例如, width=100, height=100, inSampleSize=2, 那么就会将 bitmap 处理为，width=50，height=50，宽高降为 1 / 2，像素数降为 1 / 4

inJustDecodeBounds 字面意思就可以理解就是只解析图片的边界, 有时如果只是为了获取图片的大小就可以用这个，而不必直接加载整张图片。

inPreferredConfig 默认会使用 ARGB_8888,在这个模式下一个像素点将会占用 4 个 byte,而对一些没有透明度要求或者图片质量要求不高的图片，可以使用 RGB_565，一个像素只会占用 2 个 byte，一下可以省下 50%内存。

inPurgeable 和 inInputShareable 这两个需要一起使用，BitmapFactory.java 的源码里面有注释，大致意思是表示在系统内存不足时是否可以回收这个 bitmap，有点类似软引用，但是实际在 5.0 以后这两个属性已经被忽略，因为系统认为回收后再解码实际会反而可能导致性能问题

inBitmap 官方推荐使用的参数，表示重复利用图片内存，减少内存分配，在 4.4 以前只有相同大小的图片内存区域可以复用，4.4 以后只要原有的图片比将要解码的图片大既可以复用了。

4. 缓存池大小

现在很多图片加载组件都不仅仅是使用软引用或者弱引用了，实际上类似 Glide 默认使用的事 LruCache，因为软引用 弱引用都比较难以控制，使用 LruCache 可以实现比较精细的控制，而默认缓存池设置太大了会导致浪费内存，设置小了又会导致图片经常被回收，所以需要根据每个 App 的情况，以及设备的分辨率，内存计算出一个比较合理的初始值，可以参考 Glide 的做法。

5. 内存抖动

什么是内存抖动呢？Android 里内存抖动是指内存频繁地分配和回收，而频繁的 gc 会导致卡顿，严重时还会导致 OOM。

一个很经典的案例是 string 拼接创建大量小的对象(比如在一些频繁调用的地方打字符串拼接的 log 的时候)，见 Android 优化之 String 篇[9]。

而内存抖动为什么会引起 OOM 呢？

主要原因还是有因为大量小的对象频繁创建，导致内存碎片，从而当需要分配内存时，虽然总体上还是有剩余内存可分配，而由于这些内存不连续，导致无法分配，系统直接就返回 OOM 了。

比如我们坐地铁的时候，假设你没带公交卡去坐地铁，地铁的售票机就只支持 5 元，10 元，而哪怕你这个时候身上有 1 万张 1 块的都没用(是不是觉得很反人类..)。当然你可以去兑换 5 元，10 元，而在 Android 系统里就没那么幸运了，系统会直接拒绝为你分配内存，并扔一个 OOM 给你(有人说 Android 系统并不会对 Heap 中空闲内存区域做碎片整理，待验证)。

其他

常用数据结构优化，ArrayMap 及 SparseArray 是 android 的系统 API，是专门为移动设备而定制的。用于在一定情况下取代 HashMap 而达到节省内存的目的,具体性能见 HashMap，ArrayMap，SparseArray 源码分析及性能对比[10]，对于 key 为 int 的 HashMap 尽量使用

SparceArray 替代，大概可以省 30%的内存，而对于其他类型，ArrayMap 对内存的节省实际并不明显，10%左右，但是数据量在 1000 以上时，查找速度可能会变慢。

枚举，Android 平台上枚举是比较争议的，在较早的 Android 版本，使用枚举会导致包过大，在个例子里面，使用枚举甚至比直接使用 int 包的 size 大了 10 多倍 在 stackoverflow 上也有很多的讨论，大致意思是随着虚拟机的优化，目前枚举变量在 Android 平台性能问题已经不大，而目前 Android 官方建议，使用枚举变量还是需要谨慎，因为枚举变量可能比直接用 int 多使用 2 倍的内存。

ListView 复用，这个大家都知道，getView 里尽量复用 convertView,同时因为 getView 会频繁调用，要避免频繁地生成对象

谨慎使用多进程，现在很多 App 都不是单进程，为了保活，或者提高稳定性都会进行一些进程拆分，而实际上即使是空进程也会占用内存(1M 左右)，对于使用完的进程，服务都要及时进行回收。

尽量使用系统资源，系统组件，图片甚至控件的 id

减少 view 的层级，对于可以 延迟初始化的页面，使用 viewstub

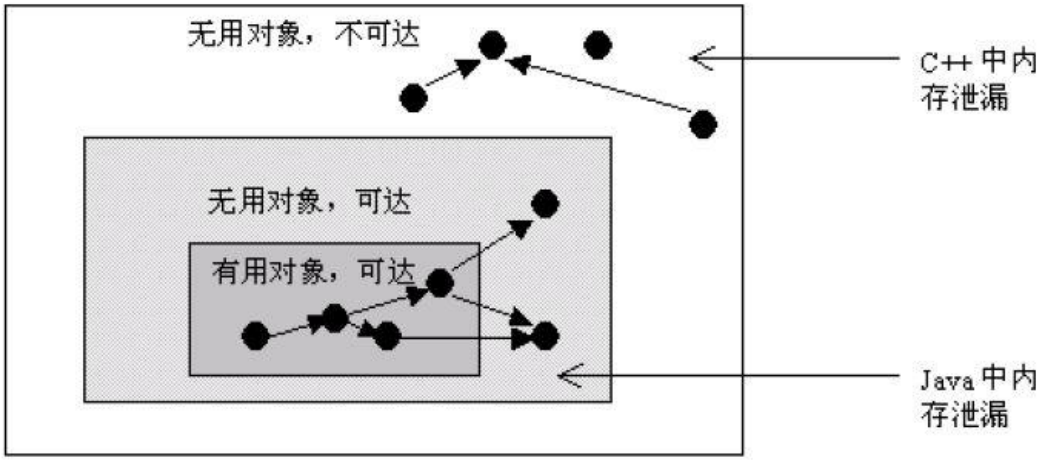
数据相关：序列化数据使用 protobuf 可以比 xml 省 30%内存，慎用 sharepreferenc，因为对于同一个 sp，会将整个 xml 文件载入内存，有时候为了读一个配置，就会将几百 k 的数据读进内存，数据库字段尽量精简，只读取所需字段。

dex 优化，代码优化，谨慎使用外部库， 有人觉得代码多少于内存没有关系，实际会有那么点关系，现在稍微大一点的项目动辄就是百万行代码以上，多 dex 也是常态，不仅占用 rom 空间，实际上运行的时候需要加载 dex 也是会占用内存的(几 M)，有时候为了使用一些库里的某个功能函数就引入了整个庞大的库，此时可以考虑抽取必要部分，开启 proguard 优化代码，使用 Facebook redex 使用优化 dex(好像有不少坑)。

Android 内存泄漏分析心得

前言

对于 C++来说，内存泄漏就是 new 出来的对象没有 delete。
对于 Java 来说，就是 new 出来的 Object 放在 Heap 上无法被 GC 回收。



本文通过 QQ 和 Qzone 中内存泄漏实例来讲 android 中内存泄漏分析解法和编写代码应注意的事项。

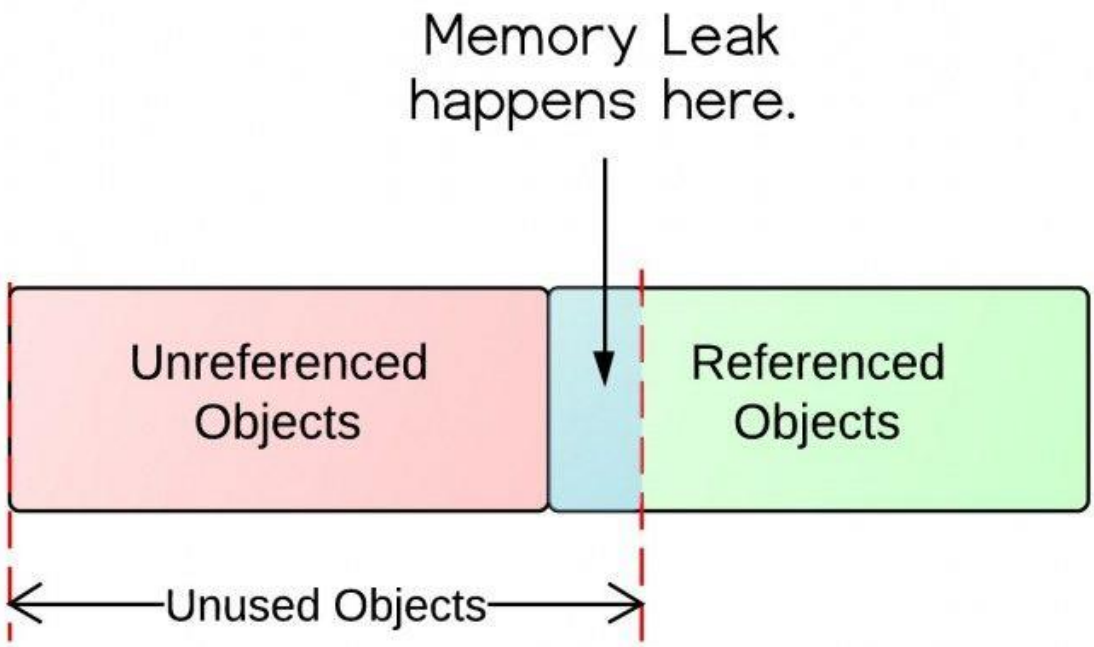
Java 中的内存分配

静态储存区：编译时就分配好，在程序整个运行期间都存在。它主要存放静态数据和常量。
栈区：当方法执行时，会在栈区内存中创建方法体内部的局部变量，方法结束后自动释放内存。
堆区：通常存放 new 出来的对象。由 Java 垃圾回收器回收。

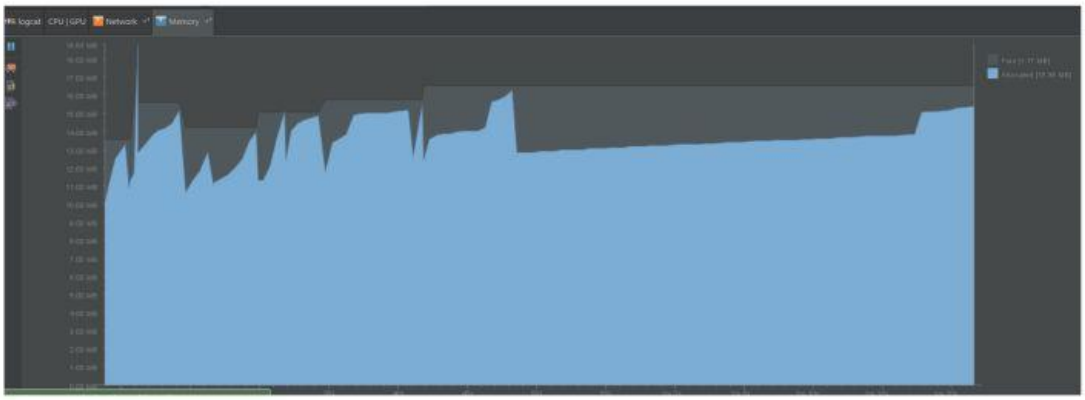
四种引用类型的介绍

强引用(StrongReference)：JVM 宁可抛出 OOM ，也不会让 GC 回收具有强引用的对象。
软引用(SoftReference)：只有在内存空间不足时，才会被回的对象。
弱引用(WeakReference)：在 GC 时，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。
虚引用(PhantomReference)：任何时候都可以被 GC 回收，当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。程序可以通过判断引用队列中是否存在该对象的虚引用，来了解这个对象是否将要被回收。可以用来作为 GC 回收 Object 的标志。

我们常说的内存泄漏是指 new 出来的 Object 无法被 GC 回收，即为强引用：



内存泄漏发生时的主要表现为内存抖动，可用内存慢慢变少：



常见的内存泄漏案例

case 1. 单例造成的内存泄露

单例的静态特性导致其生命周期同应用一样长。

解决方案：
1、将该属性的引用方式改为弱引用
2、如果传入 Context，使用 ApplicationContext

example：泄漏代码片段

```
 private static ScrollHelper mInstance;
 private ScrollHelper() {
 }
```

```
public static ScrollHelper getInstance() {
    if (mInstance == null) {
        synchronized (ScrollHelper.class) {
            if (mInstance == null) {
                mInstance = new ScrollHelper();
            }
        }
    }

    return mInstance;
}
/**
 * 被点击的 view
 */
private View mScrolledView = null;
public void setScrolledView(View scrolledView) {
    mScrolledView = scrolledView;
}
```

Solution：使用 WeakReference

```
private static ScrollHelper mInstance;
private ScrollHelper() {
}
public static ScrollHelper getInstance() {
    if (mInstance == null) {
        synchronized (ScrollHelper.class) {
            if (mInstance == null) {
                mInstance = new ScrollHelper();
            }
        }
    }

    return mInstance;
}
/**
 * 被点击的 view
 */
private WeakReference<View> mScrolledViewWeakRef = null;
public void setScrolledView(View scrolledView) {
    mScrolledViewWeakRef = new WeakReference<View>(scrolledView);
}
```

case 2. InnerClass 匿名内部类

在 Java 中，非静态内部类 和 匿名类 都会潜在的引用它们所属的外部类，但是，静态内部类却不会。如果这个非静态内部类实例做了一些耗时的操作，就会造成外围对象不会被回收，从而导致内存泄漏。

解决方案：

1、将内部类变成静态内部类

2、如果有强引用 Activity 中的属性，则将该属性的引用方式改为弱引用

3、在业务允许的情况下，当 Activity 执行 onDestory 时，结束这些耗时任务

example：

```
public class LeakAct extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.aty_leak);
        test();
    }
    //这儿发生泄漏
    public void test() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                while (true) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }).start();
    }
}
```

Solution：

```
public class LeakAct extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.aty_leak);
        test();
    }
    //加上 static，变成静态匿名内部类
    public static void test() {
```

```
        new Thread(new Runnable() {
            @Override
            public void run() {
                while (true) {
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }).start();
    }
}
```

case 3. Activity Context 的不正确使用

在 Android 应用程序中通常可以使用两种 Context 对象：Activity 和 Application。当类或方法需要 Context 对象的时候常见的做法是使用第一个作为 Context 参数。这样就意味着 View 对象对整个 Activity 保持引用，因此也就保持对 Activty 的所有的引用。
假设一个场景，当应用程序有个比较大的 Bitmap 类型的图片，每次旋转是都重新加载图片所用的时间较多。为了提高屏幕旋转是 Activity 的创建速度，最简单的方法时将这个 Bitmap 对象使用 Static 修饰。 当一个 Drawable 绑定在 View 上，实际上这个 View 对象就会成为这份 Drawable 的一个 Callback 成员变量。而静态变量的生命周期要长于 Activity。导致了当旋转屏幕时，Activity 无法被回收，而造成内存泄露。

解决方案：
1、使用 ApplicationContext 代替 ActivityContext，因为 ApplicationContext 会随着应用程序的存在而存在，而不依赖于 activity 的生命周期
2、对 Context 的引用不要超过它本身的生命周期，慎重的对 Context 使用“static”关键字。
3、Context 里如果有线程，一定要在 onDestroy()里及时停掉

example：

```
 private static Drawable sBackground;
 @Override
 protected void onCreate(Bundle state) {
     super.onCreate(state);
     TextView label = new TextView(this);
     label.setText("Leaks are bad");
     if (sBackground == null) {
         sBackground = getDrawable(R.drawable.large_bitmap);
     }
     label.setBackgroundDrawable(sBackground);
     setContentView(label);
```

```
}
```

Solution：

```
private static Drawable sBackground;
@Override
protected void onCreate(Bundle state) {
    super.onCreate(state);
    TextView label = new TextView(this);
    label.setText("Leaks are bad");
    if (sBackground == null) {
        sBackground = getApplicationContext().getDrawable(R.drawable.large_bitmap);
    }
    label.setBackgroundDrawable(sBackground);
    setContentView(label);
}
```

case 4. Handler 引起的内存泄漏

当 Handler 中有延迟的的任务或是等待执行的任务队列过长，由于消息持有对 Handler 的引用，而 Handler 又持有对其外部类的潜在引用，这条引用关系会一直保持到消息得到处理，而导致了 Activity 无法被垃圾回收器回收，而导致了内存泄露。

解决方案：

1、可以把 Handler 类放在单独的类文件中，或者使用静态内部类便可以避免泄露
2、如果想在 Handler 内部去调用所在的 Activity,那么可以在 handler 内部使用弱引用的方式去指向所在 Activity.使用 Static + WeakReference 的方式来达到断开 Handler 与 Activity 之间存在引用关系的目的。

Solution：

```
@Override
protected void doOnDestroy() {
    super.doOnDestroy();
    if (mHandler != null) {
        mHandler.removeCallbacksAndMessages(null);
    }
    mHandler = null;
    mRenderCallback = null;
}
```

case 5. 注册监听器的泄漏

系统服务可以通过 Context.getSystemService 获取，它们负责执行某些后台任务，或者为硬件访问提供接口。如果 Context 对象想要在服务内部的事件发生时被通知，那就需要把自己注册到服务的监听器中。然而，这会让服务持有 Activity 的引用，如果在 Activity onDestory

时没有释放掉引用就会内存泄漏。

解决方案：
1、使用 ApplicationContext 代替 ActivityContext
2、在 Activity 执行 onDestory 时，调用反注册

mSensorManager = (SensorManager) this.getSystemService(Context.SENSOR_SERVICE);
Solution：
mSensorManager = (SensorManager)
getApplicationContext().getSystemService(Context.SENSOR_SERVICE);

下面是容易造成内存泄漏的系统服务：
InputMethodManager imm = (InputMethodManager)
context.getApplicationContext().getSystemService(Context.INPUT_METHOD_SERVICE);

Solution：
```
protected void onDetachedFromWindow() {
    if (this.mActionShell != null) {
        this.mActionShell.setOnClickListener((OnAreaClickListener)null);
    }
    if (this.mButtonShell != null) {
        this.mButtonShell.setOnClickListener((OnAreaClickListener)null);
    }
    if (this.mCountShell != this.mCountShell) {
        this.mCountShell.setOnClickListener((OnAreaClickListener)null);
    }
    super.onDetachedFromWindow();
}
```

case 6. Cursor，Stream 没有 close，View 没有 recyle

资源性对象比如(Cursor，File 文件等)往往都用了一些缓冲，我们在不使用的时候，应该及时关闭它们，以便它们的缓冲及时回收内存。它们的缓冲不仅存在于 java 虚拟机内，还存在于 java 虚拟机外。如果我们仅仅是把它的引用设置为 null,而不关闭它们，往往会造成内存泄漏。因为有些资源性对象，比如 SQLiteCursor(在析构函数 finalize(),如果我们没有关闭它，它自己会调 close()关闭)，如果我们没有关闭它，系统在回收它时也会关闭它，但是这样的效率太低了。因此对于资源性对象在不使用的时候，应该调用它的 close()函数，将其关闭掉，然后才置为 null. 在我们的程序退出时一定要确保我们的资源性对象已经关闭。

Solution：
1、调用 onRecycled()
```
@Override
public void onRecycled() {
    reset();
```

```
        mSinglePicArea.onRecycled();
    }
     在 View 中调用 reset()
    public void reset() {
        if (mHasRecyled) {
            return;
        }
    ...
        SubAreaShell.recycle(mActionBtnShell);
        mActionBtnShell = null;
    ...
        mIsDoingAvatartRedPocketAnim = false;
        if (mAvatarArea != null) {
                mAvatarArea.reset();
        }
        if (mNickNameArea != null) {
            mNickNameArea.reset();
        }
    }
```

case 7. 集合中对象没清理造成的内存泄漏

我们通常把一些对象的引用加入到了集合容器(比如 ArrayList)中，当我们不需要该对象时，并没有把它的引用从集合中清理掉，这样这个集合就会越来越大。如果这个集合是 static 的话，那情况就更严重了。
所以要在退出程序之前，将集合里的东西 clear，然后置为 null，再退出程序。

解决方案：
在 Activity 退出之前，将集合里的东西 clear，然后置为 null，再退出程序。

Solution

```
    private List<EmotionPanelInfo> data;
    public void onDestory() {
        if (data != null) {
            data.clear();
            data = null;
        }
    }
```

case 8. WebView 造成的泄露

当我们不要使用 WebView 对象时，应该调用它的 destory()函数来销毁它，并释放其占用的内存，否则其占用的内存长期也不能被回收，从而造成内存泄露。

解决方案：
为 webView 开启另外一个进程，通过 AIDL 与主线程进行通信，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，从而达到内存的完整释放。

case 9. 构造 Adapter 时，没有使用缓存的 ConvertView

初始时 ListView 会从 Adapter 中根据当前的屏幕布局实例化一定数量的 View 对象，同时 ListView 会将这些 View 对象 缓存起来。

当向上滚动 ListView 时，原先位于最上面的 List Item 的 View 对象会被回收，然后被用来构造新出现的最下面的 List Item。

这个构造过程就是由 getView()方法完成的，getView()的第二个形参 View ConvertView 就是被缓存起来的 List Item 的 View 对象(初始化时缓存中没有 View 对象则 ConvertView 是 null)。