> 介绍性能优化相关方面的优化。

# Android性能优化详解

版本：2018/8/9-1(15:46)



## 性能问题分类

1、性能问题分类

> 1. 渲染性能问题：界面卡顿
> 2. 代码性能问题：数据结构高效、ANR
> 3. 内存问题：内存浪费、内存泄露
> 4. 网络问题：重复请求
> 5. 功耗问题：耗电

2、优化方法

> 1. 了解问题：滑动卡顿等明显的问题可以快速定位，难以察觉的问题需要借助工具。
> 2. 定位问题：通过工具检测、逻辑分析、数据分析，对问题进行定位。
> 3. 分析问题：分析针对这个问题该如何解决，确定解决方案。
> 4. 解决问题：根据解决方案进行优化。
> 5. 验证问题：对优化结果进行验证，此外验证没有产生其他问题。

# 性能优化指标

1、渲染

- 滑动流畅度：FPS，即Frame per Second，一秒内的刷新帧数，越接近60帧越好；
- 过度绘制：单页面的3X（粉红色区域） Overdraw小于25%
- 启动时间：这里主要说的是Activity界面启动时间，一般低于300ms，需要用高频摄像机计算时间。

2、内存

- 内存大小：峰值越低越好，需要优化前后做对比
- 内存泄漏：需要用工具检查对比优化前后

3、代码

- 数据结构：在合适场景采用合适的集合。
- 性能：越快越好---在各种场景下采用最优方案(避免ANR、避免使用性能低下API等)。

4、功耗

- 单位时间内的掉电量越少越好。

# 一、渲染优化(34)

## 基础知识

1、UI渲染机制

1. 系统通过VSYNC信号触发对UI的渲染、重绘，其间隔时间是16ms。这个16ms其实就是1000ms显示60帧画面的单位时间。
2. 如果不能在16ms内完成绘制，那么就会造成丢帧现象。会在下个信号才开始绘制（2*16ms）---导致卡顿

2、UI渲染时间工具

1. 开发者选项，选择 `Profile GPU Rendering`，并选中 `On screen as bars` 的选项，这时候显示一些条形图。
2. 蓝色: 绘制Display List时间
3. 红色: OpenGL渲染Display List所需要的时间
4. 黄色: CPU等待GPU处理的时间
5. 需要尽量都控制在 绿线 之下

3、View是什么?

1. View是Android系统在屏幕上的视觉呈现.
2. View是一种控件

4、View是怎么绘制出来的?

1. View的绘制流程是从ViewRoot的performTraversals（）方法开始
2. measure()
3. layout()
4. draw()

5、View是怎么呈现在界面上的?

1. Android中的视图都是通过Window来呈现的(Activity、Dialog还是Toast都有一个Window)
2. 然后通过WindowManager来管理View。
3. Window和顶级View——DecorView的通信是依赖ViewRoot完成的。

6、View和ViewGroup什么区别?

1. View是所有控件和布局的父类。
2. `ViewGroup` 也属于 `View`，构成一种树状结构

7、视图优化的几种方法

1. 减少画面绘制的时间
2. 避免过度绘制
3. 优化布局层级
4. 避免嵌套过多无用布局

# 过度绘制

8、过度绘制是什么?

> 1. 一个像素重复绘制了多次
> 2. 例如：先绘制Activity的背景，再给布局绘制了重叠的背景。

9、过度绘制的检查工具

> 检查工具 `Enable GPU OverDraw`

10、Activity中去除主题的背景

> 1. 在MainActivity的Theme中修改背景，去除布局（main_activity_layout.xml）中的background。
> 2. 如果不给当前Activity设置主题，会使用默认主题的背景，需要主动添加主题并且去除背景。

```
//styles.xml
<style name="AppBaseTheme" parent="android:Theme.Light">
    <item name="android:windowBackground">@null</item>
</style>
```

11、移除Window的默认背景

```
//Activity中
getWindow().setBackgroundDrawable(null);
```

12、将底层Fragment设置透明背景

```
android:background="#00000000"
```

13、移除布局控件的多余背景

```
android:background="@null"
```

14、去除代码中添加的多余背景

> 1. 除了布局中多余背景，还有可能在代码里添加了多余的背景。
> 2. 例如在弹窗中的Window里需要将背景设置为null

15、Dialog的过度绘制问题

> 1. 如果采用系统dialog，不需要考虑过度绘制问题，因为其弹窗绘制是属于剪切式绘制不是覆盖绘制。
> 2. 如果自定义一个弹窗view，就需要考虑过度绘制问题。

16、自定义view中控制绘制范围

1. 自定义View时，使用clipRec控制绘制范围
2. Canvas的clipRect方法控制每个视图每次刷新的区域，这样可以避免刷新不必要的区域，从而规避过渡绘制的问题。
3. 还可以使用canvas.quickreject()来判断是否和某个矩形相交，从而跳过那些非矩形区域内的绘制操作。

# 布局的选择

17、布局优化的思想

1. 减少布局文件层级
2. 使用高性能布局

18、RelativeLayout和LinearLayout的性能差异

1. 两者 layout和draw 性能相等，区别在于 measure 过程
2. RelativeLayout 对所有 子View 会进行 两次measure(横向+纵向) ---因为 子View 间可能同时有 纵向和横向的依赖关系 ， 所有都需要进行一次测量。
3. LinearLayout 会进行 一次measure ， 如果有 weight属性 才会进行 第二次measure

19、布局的选择

1. 如果不涉及到 层级深度 ， 应该选择 高效的LinearLayout或者FrameLayout
2. 涉及到层级深度时，如 ListView 中更适合使用 RelativeLayout ， 且尽可能使用 pading 代替 margin
3. ConstraintLayout 是性能最好的布局!

# 优化布局层级

20、布局层级是什么?

1. 降低View树的高度
2. 对View的测量、布局、绘制，都是对View树进行遍历。
3. API文档建议View树高度不宜超过10层。
4. 谷歌将XML文件默认根布局从 LinearLayout 替换成了 RelativeLayout ， 就是避免前者嵌套所产生布局树的高度，从而提高UI渲染的效率。

21、如何查看布局层级

1. 通过Android Studio中Tools内的 Layout Inspector
2. 老版本是 Hierarchy Viewer

22、布局层级优化的三种方法

1. include 标签
2. merge 标签

3. ViewStub

## include标签

23、include标签的特点和使用方法

1. include 主要是用于布局重用，就不需要重复写相同的布局。(将类似的内容来定义一个通用UI)
2. include 中只支持 layout_ 开头的属性，如果指定了 id 则会覆盖掉 所包含的布局文件根元素的id属性
   Tips: 将 height width 设置为 0dp ,迫使开发者对宽高进行重新设置（否则看不见）

```
//main.xml
<android.support.constraint.ConstraintLayout ...>
    <include
        android:id="@+id/include"
        layout="@layout/my_include_layout"
        android:layout_width="200dp"
        android:layout_height="200dp"
        .../>
    <TextView
        android:id="@+id/sample_text"
        ... />
</android.support.constraint.ConstraintLayout>

//my_include_layout.xml
<android.support.constraint.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/sample_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        .../>


</android.support.constraint.ConstraintLayout>
```

24、include标签的注意点

1. 若需要覆盖原来原来布局的类似 layout_xxxx 的属性.需要在 include 中同时指定 layout_width 和 layout_height 属性

## merge标签

25、merge标签的作用

1. merge 一般配合 include标签 使用，用于减少 布局层级
2. 假如 include标签 外层是 LinearLayout ， 内部布局根元素的布局也是 LinearLayout ， 使用 merge 能减少不必要的层级。

```
//需要将include所包含的内部布局的根布局使用merge标签，如my_include_layout.xml
<merge xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <TextView
        android:id="@+id/sample_text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="I'm include"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</merge>
```

## ViewStub标签

26、ViewStub的作用和使用

1. 使用<ViewStub>实现View的延迟加载
2. 是一种轻量级组件(继承View)：不可视、大小为0
3. ViewStub 一旦进行显示，就不存在了，取而代之的是被inflate的布局，并将ID设置为ViewStub中 android：inflateid 属性所指的ID，因此两次调用 inflate 方法会报错
4. ViewStub 还不支持 merge标签

```
<ViewStub
 android:id="@+id/not_often_use"
 ......
 android:layout="@layout/not_often_use"/>


//1. 获得ViewStub组件
mViewStub = (ViewStub)findViewById(R.id.not_often_use); //viewstub的ID
//2-1. 显示该View的第一种方法
mViewStub.setVisibility(View.VISIBLE)
//2-2. 显示该View的第二种方法
View flateView = mViewStub.inflate();
TextView textview = (TextView)inflateView.findViewById(R.id.tv);
```

27、如何获取VieStub组件

```
mViewStub = (ViewStub)findViewById(R.id.not_often_use); //viewstub的ID
```

28、如何显示ViewStub的内容(两种方法)

1. `mViewStub.setVisibility(View.VISIBLE)`
2. 调用ViewStub的inflate并且获取到实际的View控件

```
View inflateView = mViewStub.inflate();
TextView textview = (TextView)inflateView.findViewById(R.id.tv);
```

29、ViewStub显示的注意点

> 1. 不管是哪种方式，在显示后， `ViewStub` 就不存在了，取而代之的是被 `inflate` 的 `Layout` 。
> 2. 并将ID重新设置为ViewStub中 `android：inflateid`属性 所指的ID。
> 3. 因此 两次调用`inflate`方法会报错

30、ViewStub和View.Gone的区别?

> 1. ViewStub只会在显示的时候采会渲染布局(ViewStub更有效率)
> 2. View.Gone 初始化布局树的时候已经添加在布局树上

# 绘制优化

31、绘制优化-优化View的onDraw方法

> 1. `onDraw` 中不要创建新的 局部对象 (内存分配)--- `onDraw` 可能会频繁调用，瞬间创建大量对象和大量GC
> 2. `onDraw` 中不要做耗时的任务，尽量要保证 `View`的绘制频率`60fps(`每帧画面不超过`16ms)`

32、使用RenderScript、OpenGL进行复杂的绘图操作。

33、使用SurfaceView代替View进行大量、频繁的绘图操作。

34、尽量使用视图缓存，而不是每次都执行inflate()方法解析视图。

# 二、代码优化(16)

1、HashMap和hashtable

> 1. HashMap的性能比Hashtable高很多，不要采用后者。(Hashtable大量使用%而不是位运算，性能差)
> 2. ConcurrentHashMap的性能比Hshtable高很多。(Hashtable直接是大量synchronized方法，效率低下。前者使用双层哈希表、Segment和锁分段技术，因此效率很高。)

2、HashMap性能比TreeMap要高.(TreeMap内部是红黑树实现，但是没有HashMap性能高)

3、HashSet总体性能比TreeSet要高。TreeSet的迭代上比HashSet性能更高。或者在有序的情况下，TreeSet的性能也会高一些。

4、SparseArray在一定场景下比HashMap更好。

> 1. 在Key=Integer value=Object时，采用SparseArray比HashMap效率更高(可以免去 基本对象类型 转换为 引用类型 所用的时间。)

5、Pair的使用

> 1. 可以满足一种场景：既需要以键值的方式存储数据列表，还需要在输出的时候保持顺序。(HashMap满足key-value， ArrayList满足顺序)
> 2. Pair + ArrayList 进行使用。

6、Handler避免内存泄露

> 1. 使用静态的内部类，静态内部类不会持有外部类的引用。
> 2. 并用弱引用 `WeakReference` 进行处理。

7、Context的正确使用(单例模式中必须要Context的解决办法)。

> 1. 使用Activity、Service的Context会导致单例一直持有应用，从而内存泄露。
> 2. 如果单例中一定要使用到Context对象，建议使用Application的Context

8、不要使用Enum枚举

> 1. Android官方强烈建议不要在Android程序里面使用到enum。
> 2. 例如使用static int的代码编译成dex的大小是2680 bytes。使用enum之后的dex大小是4188 bytes，相比起2556增加了1632 bytes，增长量是使用static int的13倍。
> 3. 不仅仅如此，使用enum，运行时还会产生额外的内存占用。

9、 减少不必要的对象.

> 1. 应该避免频繁创建短作用域的变量。
> 2. 避免在For循环中创建临时对象，要减少GC的次数。

10、多使用静态方法，静态方法会比普通方法提高15%的访问速度，也可以避免创建额外对象。

11、尽量使用 基本类型 代替 引用类型

12、多使用 系统API ，如 `System.arrayCopy()` 进行数组拷贝，比for循环的效率快9倍以上。

13、减少不必要的成员变量，这点在Android Lint工具上已经集成检测了，如果一个变量可以定义为局部变量，则会建议你不要定义成成员变量。

14、资源要及时释放

> 对Cursor、Receiver、Sensor、File等对象，要非常注意对它们的创建、回收与注册、解注册

15、避免使用IOC框架

1. IOC(Inversion of Control)---控制反转。
2. IOC通常使用注解、反射来实现，虽然Java已经进行了很好的优化，但大量使用反射依然会带来性能的下降。
3. ButterKnife、Android Annotations都是IOC框架。

16、常量使用 `static final` 进行修饰

1. 对于基本类型和String类型的常量，建议使用常量static final 修饰
2. final类型的常量会在静态dex文件的域初始化部分
3. 对这些常量的调用不会涉及类的初始化，而是直接调用字面量。

17、适当使用 软引用 和 弱引用

18、尽量使用静态内部类，避免由于内部类导致的内存泄露

19、减少不必要的成员变量

在Android Lint工具上已经集成检测了，如果一个变量可以定义为局部变量，则会建议你不要定义成成员变量。

# 三、内存优化(13)

## 基础知识

1、Android的内存机制

1. android 沙箱机制 ，每个应用给定内存。过多内存会触发 `LMK - Low Memory Killer`

2、内存是什么?

1. 内存就是RAM

3、RAM的组成

1. Register：
2. stack：存放 基本数据类型 和 对象的引用 .但对象本身不存在stack，而是存放在 堆 中.
3. heap：存放new创建的对象和数据。由** `GC` -Java虚拟机的垃圾回收器**
4. static field：固定的位置存放应用程序运行时一直存在的数据，java中管理一些静态的数据变量
5. constant pool：java虚拟机必须为每个被装载的类型维护一个常量池。常量池就是该类型所用到常量的有序集合，包括直接常量（基本类型，String）和对其他类型、字段、方法的符号引用。

4、stack栈和heap堆的区别

1. stack分配的内存空间会在该变量作用域结束后，这部分内存立即释放。

2. heap为 new 创建的变量分配的空间，不会立即释放，而是等待系统 GC 进行回收。

5、如何获得堆的大小

```
ActivityManager manager = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
        int heapsize= manager.getLargeMemoryClass();
```

6、Java如何进行内存回收

1. java创建 垃圾收集器线程(Garbage Collection Thread) 进行资源管理。
2. 调用 System.gc 能建议系统进行GC，但是不一定GC。
3. 内存泄漏的原因：再强大的算法也会出现部分对象忘记回收的现象。

7、ANR是什么?

1. UI线程执行耗时操作会导致ANR
2. 如果 BroadcastReceiver 10秒没有执行完操作，就会导致ANR

8、ANR如何分析?

1. 只要出现 ANR ， 系统都会在 /data/anr/目录 下生成相应的 trace.txt
2. 根据内容可以分析 ANR 的具体原因

9、ANR出现的几种特殊情况

1. 主线程UI线程 中进行 耗时操作
2. UI线程 和在其他线程的 耗时操作 竞争同一个锁。

# 内存泄露优化

10、内存泄漏优化

1. 静态变量 所导致的内存泄漏：如Activity内部静态变量持有Activity的this等
2. 单例模式 导致的内存泄漏：单例模式的对象中的链表等持有了 如：Activity的this指针 ， 却没有即使释放会导致泄露，因为单例的特点是其生命周期和 Applicaiton 一致。
3. 属性动画 不停止会导致内存泄漏：无限循环动画会持有 Activity的View ， 而 View 持有了 Activity ， 最终导致泄露。解决办法是在 onDestory 中调用 animator.cancel()

# 列表优化

11、ListView/GridView的优化

1. 采用 ViewHolder
2. 避免在 getView 中执行耗时操作
3. 根据滑动状态控制任务的 执行频率 ， 避免快速滑动时开启大量的异步操作。
4. 开启硬件加速，使列表滑动更流畅。

## bitmap优化

12、Bitmap的优化

> 1. bitmap 是内存占用过高，甚至 OOM(out of memory) 的最大威胁。
> 2. 选择合适大小图片，也可以在图片列表时显示缩略图，显示图片的时候使用原图
> 3. 通过 BitmapFactory.Options 对图片进行采样、
> 4. 图片缓存-使用 内存缓存LruCache 和 硬盘缓存DiskLruCache
> 5. 可以直接使用成熟的 Glide、Fresco 进行图片显示。

## 线程优化

13、线程优化

> 1. 采用线程池，避免线程的创建和销毁带来的性能损失
> 2. 线程池能有效控制最大并发数，避免抢占资源导致的阻塞

# 四、功耗优化(3)

1、功耗优化

> 本质就是减少App使用和在后台的电量消耗。

2、节制使用Service

> 1. 节制使用 Service ，系统总是倾向于保留 Service 所依赖的进程，会造成系统资源浪费。可以使用 IntentService ，在任务执行完后，会自动停止。

3、如何让Toast在应用退出后不再显示?

> 1-在显示 Toast 信息时，判断应用当前是否在任务栈的栈顶。

```
// 1-用于判断应用是否在前台。
public static boolean isActivityRunning(String packagename, Context context){
     ActivityManager am = (ActivityManager)context.getSystemService(Context.ACTIVITY_SERVICE);
     List<RunningTaskInfo> runningTaskInfos = am.getRunningTasks(1);
     String cmpNameTemp = null;
     if(null != runningTaskInfos){
         cmpNameTemp = runningTaskInfos.get(0).topActivity.toString();
     }
     if(null != cmpNameTemp){
         return cmpNameTemp.contains(packagename);
     }
     return false;
}
```

> 2-对Toast进行封装处理

```
public static void showToast(Context context, String text, int duration) {
    if (mToast != null)
        mToast.setText(text);
    else
        mToast = Toast.makeText(context, text, Toast.LENGTH_SHORT);

    if(isActivityRunning(context.getPackageName(), context)){
         mToast.show();
    }
}
```

# 五、网络优化(1)

1、对常用的网络请求进行三级缓存

> 1. 发起网络请求时，先去内存中寻找缓存(LruCache)
> 2. 内存中没有缓存时，到磁盘缓存上取(DiskLruCache)
> 3. 没有找到再发起网络请求

# 六、性能分析和优化工具

1、性能优化工具汇总

> 1. UI渲染时间工具：开发者中选择 `Profile GPU Rendering(GPU呈现模式分析)` ，选择 `On Screen as bars` 能开启条形图。 蓝色线条 为绘制的时间，要控制在绿线之下。
> 2. 过度绘制：开发者中 `Enable GPU OverDraw(开启过度绘制检查)`
> 3. 布局层级： `Hierarchy Viewer` ，在 `Android Device Monitor` 中使用。
> 4. 代码提示工具：Lint
> 5. Memory Monitor
> 6. TraceView日志：分析性能问题
> 7. MAT：分析内存的强力助手
> 8. Dumpsys
> 9. Memory Info: 系统上的内存监视工具

## 1-Lint

2、Lint是什么?

> 1. Google提供的代码提示工具。用于减少代码隐藏风险，对代码习惯帮助很大。而且还能清除无用的文件等内容。
> 2. 使用方法：Analyze->Run inspection by name > unused resources=去除无用资源

# 2-Memory Monitor

3、Memory Monitor是什么?

> 1. 内存监视工具:
> 2. 内存持续升高可能发生内存泄露
> 3. 内存突然降低，可能GC

# 3-TraceView日志

4、TraceView日志的作用?

> 1. 可以分析一些性能问题，比如app中有些列表在滑动的时候会有卡顿现象。

5、如何通过代码精确生成traceview日志

> 1-在manifest中增加WRITE_EXTERNAL权限
> 2-在需要监控的代码onCreate和onDestory中添加代码

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_qqside_menu);

    //开启TraceView监听
    Debug.startMethodTracing("feather"); //制定名字如feather.trace
}

protected void onDestroy() {
    super.onDestroy();
            //结束监听
    Debug.stopMethodTracing();
}
```

> 3-复制日志代码到本地
> `adb pull /sdcard/feather.trace C:\AFeatherTool`

6、如何通过Android Device Monitor生成TraceView日志

> 1. 打开AS的Android DEVICE Monitor工具，选择调试的进程，点击工具栏中的“start method profiling”。
> 2. 有两种监听方式:
>    1. 整体监听：追踪每个方法执行的全部过程，资源消耗大
>    2. 抽样监听：按照一定频率采样，这种方法需要执行较长的时间来获取准确的数据。

7、如何打开并且分析traceview日志

> 1. 打开日志： sdk的tools\traceview.bat工具来打开。或者ADM openfile来打开日志文件。
> 2. 分析日志: 上方是用于显示方法执行时间的时间轴区域。下方是显示详细信息的profile区域。

    1. 时间轴区域：显示了不同线程在不同时间段内的执行情况。每一行都是一个独立线程。
    2. profile：所选方法执行期间的性能分析。Incl CPU time-占用CPU的时间。Excl CPU TIME-方法本身不包括子方法的占用CPU的时间。Incl/Excl real time-同理，是真正执行的时间。calls+RecurCalls-掉用次数+递归回调的次数。
3. 如果占用时间长，calls+RecurCalls次数少的需要多关注。

## 4-MAT

8、MAT是什么?

1. 用于分析APP内存状态
2. MAT(Memory analyzer tool)是分析内存的强力助手。需要下载。

9、生成HPROF文件的工具

1. 打开ADM
2. 选择要监听的线程
3. 菜单栏"update heap"
4. heap的标签中选择Cause GC就会显示当前内存状态。
5. 选择“Dump HPROF File”按钮，会生成xxx.hprof文件。
6. 在SDK的platform-tools中用 `hprof-conv C:\xxxx.hprof heap.hprof` 之后才可以用来分析
7. MAT打开即可分析

10、MAT检查内存泄露的技巧

不停点击cause GC，如果total size有明显变化就可能存在内存泄露。

## 5-Dumpsys命令

11、Dumpsys命令的作用

1. 使用Dumpsys命令分析系统状态
2. Dumpsys可以列出android系统相关的信息和服务状态。
3. 使用方法： `adb shell dumpsys + 各种参数` 。
4. Linux下配合shell命令grep、find等功能更加强大

# 参考资料


1. ViewStub
2. Android性能优化《Android群英传》第十章
3. RelativeLayout和LinearLayout的性能对比
4. 【腾讯优测干货分享】安卓专项测试之GPU测试探索
5. Android开发者选项——Gpu呈现模式分析
6. 如何让Toast消息在应用退出后不再显示

7. Android性能全面分析与优化方案
8. 安卓代码、图片、布局、网络和电量优化
9. Android主流IOC框架