Android 性能优化最佳实践

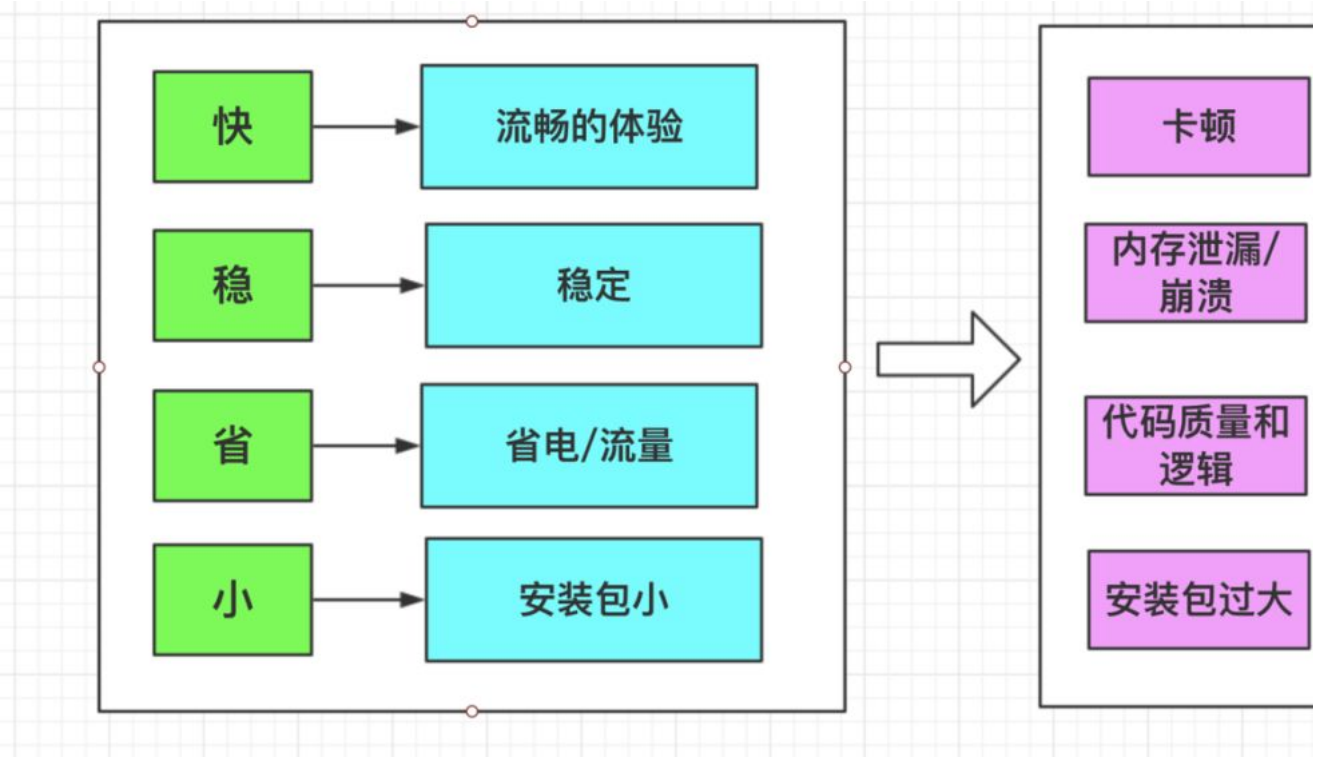
原创： 任玉刚

本文由玉刚说写作平台提供写作赞助

赞助金额：200元

原作者：Mr.S

什么是性能



这张图很好诠释了什么性能

快，稳，省，小，这四点很形象的代表了性能的四个方面，同时也让我们知道我们App现在是否是款性能良好的APP，如果有一项不达标，那么说明我们的应用有待优化。

很多时候我们注重功能实现，保证能用，但是我们会发现，这样的应用很难拿的出手，里面的槽点太多了，性能很差，但是又不知道从哪里下手进行优化，那么我们就一步一步来，看看我们到底应该怎么优化我们的APP。

1 、布局优化

和UI相关的首先就是布局，特别是在开发一些复杂界面的时候，通常我们都是采用布局嵌套的方法，每个人的布局思路不太一样，写出的也不太一样，所以就可能造成嵌套的层级过多。

官方

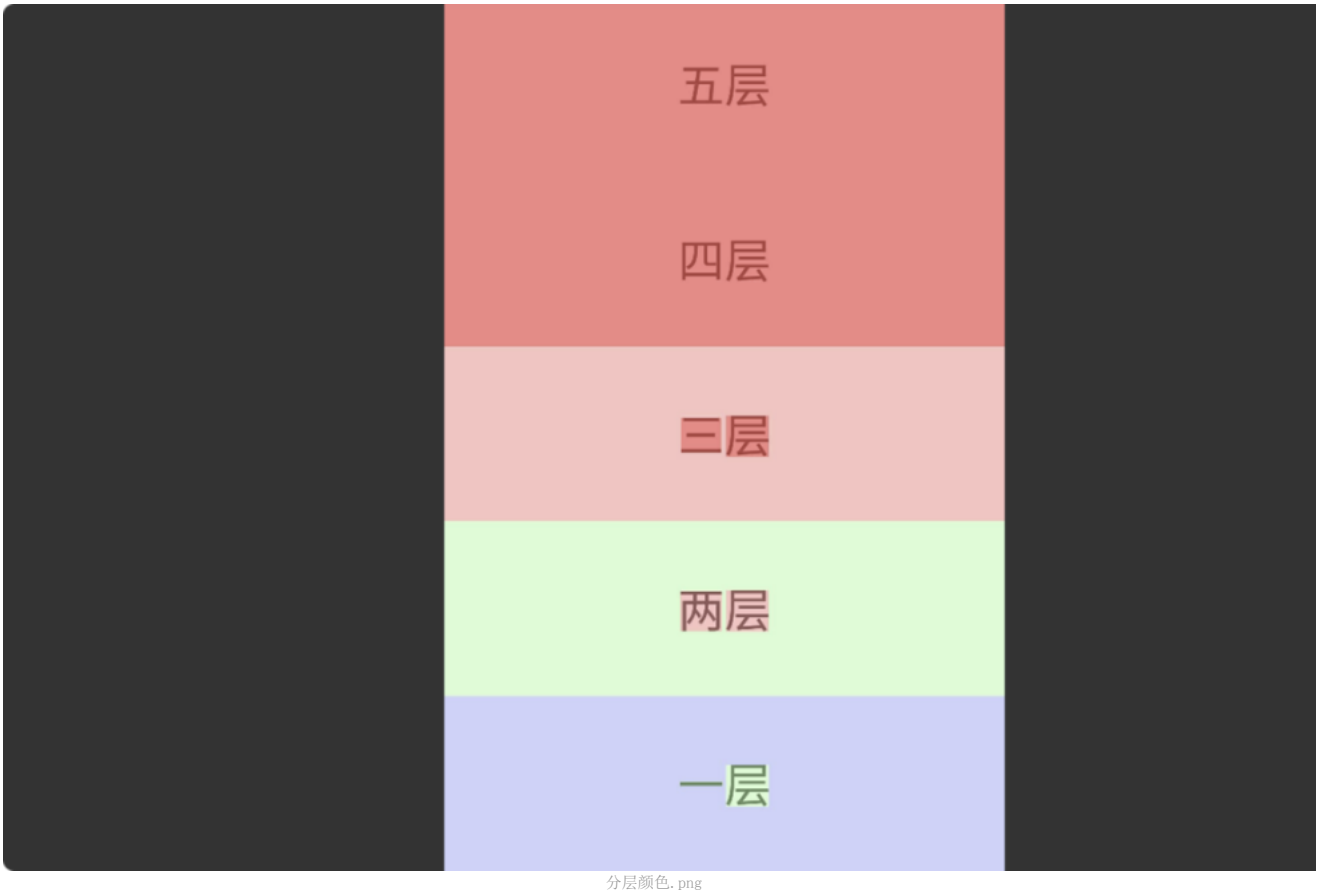
屏幕上的某个像素在同一帧的时间内被绘制了多次。在多层次的UI结构里面，如果不可见的UI也在做绘制的操作，这就会导致某些像素区域被绘制了多次。这就浪费大量的CPU以及GPU资源。

白话

显示一个布局就好比我们盖一个房子，首先我们要测量房子的大小，还要测量房间里面各个家具的大小，和位置，然后进行摆放同时也要对房子进行装修，如果我们是一层，都在明面上，干起活来敞亮也轻松，可是有的人的房子，喜欢各种隔断，分成一个一个的大隔断间，每个大隔断间里还有小隔断间，小隔断间里有小小隔断间，还有小小小隔断间。。。N层隔断间。

看到这些头皮发麻吧，而且是一个大隔断间里面所有的小隔断，小小隔断等等都测量完摆放好，才能换另外一个大隔断，天呢，太浪费时间了，不能都直接都放外面吗？也好摆放啊，这么搞我怎么摆，每个隔断间都要装修一遍，太浪费时间了啊。

我们的Android虚拟机也会这么抱怨，咱们家本来就不富裕，什么都要省着用，你这么搞，肯定运转有问题啊，那么多嵌套的小隔断间需要处理，都会占用cpu计算的时间和GPU渲染的时间。显示GPU过度绘制，分层如下如所示：



分层颜色.png

通过颜色我们可以知道我们应用是否有多余层次的绘制，如果一路飘红，那么我们就要相应的处理了。

所以我们有了第一个优化版本：

优化 1.0

1. 如果父控件有颜色，也是自己需要的颜色，那么就不必在子控件加背景颜色
2. 如果每个自控件的颜色不太一样，而且可以完全覆盖父控件，那么就不需要再父控件上加背景颜色
3. 尽量减少不必要的嵌套
4. 能用LinearLayout和FrameLayout，就不要用RelativeLayout，因为RelativeLayout控件相对比较复杂，测绘也想要耗时。

做到了以上4点只能说恭喜你，入门级优化已经实现了。

针对嵌套布局，谷歌也是陆续出了一些新的方案。对就是include、merge和ViewStub三兄弟。

include可以提高布局的复用性，大大方便我们的开发，有人说这个没有减少布局的嵌套吧，对，include确实没有，但是include和merge联手搭配，效果那是杠杠滴。

merge的布局取决于父控件是哪个布局，使用merge相当于减少了自身的一层布局，直接采用父include的布局，当然直接在父布局里面使用意义不大，所以会和include配合使用，既增加了布局的复用性，用减少了一层布局嵌套。

ViewStub它可以按需加载，什么意思？用到他的时候喊他一下，再来加载，不需要的时候像空气一样，在一边静静的呆着，不吃你的米，也不花你家的钱。等需要的时候ViewStub中的布局才加载到内存，多节俭持家啊。对于一些进度条，提示信息等等八百年才用一次的功能，使用ViewStub是极其合适的。这就是不用不知道，一用戒不了。

我们开始进化我们的优化

优化 1.1

1. 使用include和merge增加复用，减少层级
2. ViewStub按需加载，更加轻便

可能又有人说了：背景复用了，嵌套已经很精简了，再精简就实现了不了复杂视图了，可是还是一路飘红，这个怎么办？面对这个问题谷歌给了我们一个新的布局ConstraintLayout。

ConstraintLayout可以有效地解决布局嵌套过多的问题。ConstraintLayout使用约束的方式来指定各个控件的位置和关系的，它有点类似于 RelativeLayout，但远比RelativeLayout要更强大(照抄隔壁IOS的约束布局)。所以简单布局简单处理，复杂布局ConstraintLayout很好使，提升性能从布局做起。

再次进化：

优化 1.2

1. 复杂界面可选择ConstraintLayout，可有效减少层级

2、绘制优化

我们把布局优化了，但是和布局息息相关的还有绘制，这是直接影响显示的两个根本因素。

其实布局优化了对于性能提升影响不算很大，但是是我们最容易下手，最直接接触的优化，所以不管能提升多少，哪怕只有百分之一的提升，我们也要做，因为影响性能的地方太多了，每个部分都提升一点，我们应用就可以提升很多了。

我们平时感觉的卡顿问题最主要的原因之一是因为渲染性能，因为越来越复杂的界面交互，其中可能添加了动画，或者图片等等。我们希望创造出越来越炫的交互界面，同时也希望他可以流畅显示，但是往往卡顿就发生在这里。

这个是Android的渲染机制造成的，Android系统每隔16ms发出VSYNC信号，触发对UI进行渲染，但是渲染未必成功，如果成功了那么代表一切顺利，但是失败了可能就要延误时间，或者直接跳过去，给人视觉上的表现，就是要么卡了一会，要么跳帧。

View的绘制频率保证60fps是最佳的，这就要求每帧绘制时间不超过16ms(16ms = 1000/60)，虽然程序很难保证16ms这个时间，但是尽量降低onDraw方法中的复杂度总是切实有效的。

这个正常情况下，每隔16ms draw()一下，很整齐，很流畅，很完美。

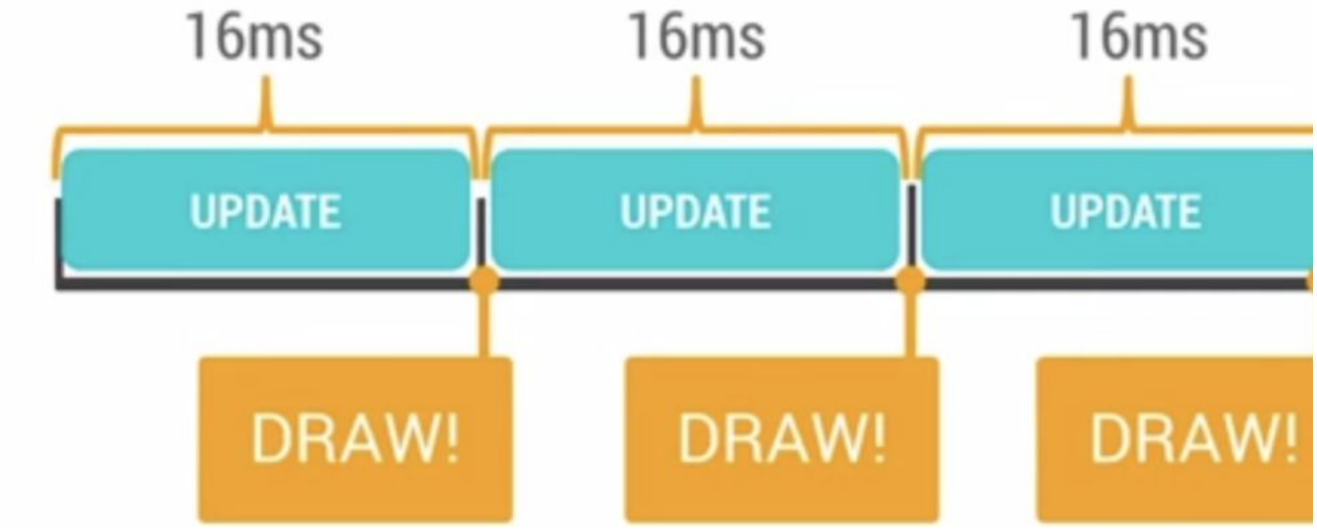


image.png

往往会发生如下图的情况，有个便秘的家伙霸占着，一帧画面拉的时间那么长，这一下可不就卡顿了嘛。把后面的时间给占用了，后面只能延后，或者直接略过了。

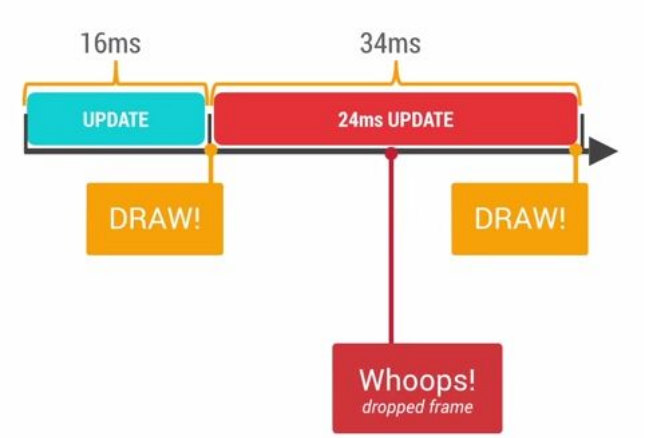


image.png

既然问题找到了，那么我们肯定要有相应的解决办法，根本做法是 减轻onDraw()的负担。

所以

第一点：

onDraw方法中不要做耗时的任务，也不做过多的循环操作，特别是嵌套循环，虽然每次循环耗时很小，但是大量的循环势必霸占CPU的时间片，从而造成View的绘制过程不流畅。

第二点：

除了循环之外，onDraw()中不要创建新的局部对象，因为onDraw()方法一般都会频繁大量调用，就意味着会产生大量的零时对象，不进占用过的内存，而且会导致系统更加频繁的GC，大大降低程序的执行速度和效率。

其实这两点在android的UI线程中都适用。

升级进化：

优化2.0

1. onDraw中不要创建新的局部对象
2. onDraw方法中不要做耗时的任务

其实从渲染优化里我们也牵扯出了另一个优化，那就是内存优化。

3、内存优化

内存泄漏指的是那些程序不再使用的对象无法被GC识别，这样就导致这个对象一直留在内存当中，占用了没来就不多的内存空间。

内存泄漏是一个缓慢积累的过程，一点一点的给你，温水煮青蛙一般，我们往往很难直观的看到，只能最后内存不够用了，程序奔溃了，才知道里面有大量的泄漏，但是到底是那些地方？估计是狼烟遍地，千疮百孔，都不知道如何下手。怎么办？最让人难受的是内存泄漏情况那么多，记不住，理解也不容易，关键是老会忘记。怎么办呢？老这么下去也不是事，总不能面试的时候突击，做项目的时候不知所措吧。所以一定要记住了解GC原理，这样才可以更准确的理解内存泄漏的场景和原因。不懂GC原理的可以先看一下这个JVM初探：内存分配、GC原理与垃圾收集器。

本来GC的诞生是为了让java程序员更加轻松（这一点隔壁C++痛苦的一匹），java虚拟机会自动帮助我们回收那些不再需要的内存空间。通过引用计数法，可达性分析法等等方法，确认该对象是否没有引用，是否可以被回收。

有人会说真么强悍的功能看起来无懈可击啊，对，理论上可以达到消除内存泄漏，但是很多人不按常理出牌啊，往往很多时候，有的对象还保持着引用，但逻辑上已经不会再用到。就是这一类对象，游走于GC法律的边缘，我没用了，但是你又不知道我没用了，就是这么赖着不走，空耗内存。

因为有内存泄漏，所以内存被占用越来越多，那么GC会更容易被触发，GC会越来越频发，但是当GC的时候所有的线程都是暂停状态的，需要处理的对象数量越多耗时越长，所以这也会造成卡顿。

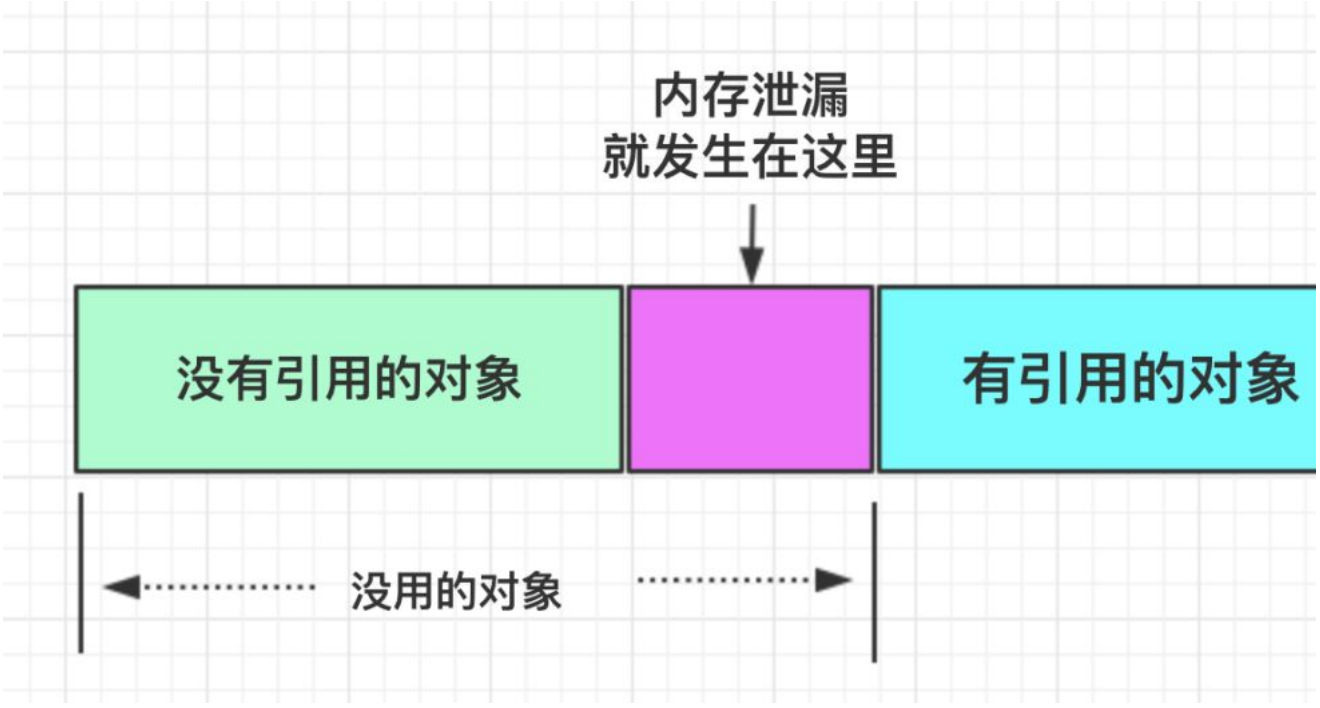


image.png

那么什么情况下会出现这样的对象呢？

基本可以分为以下四大类：

1、集合类泄漏

2、单例/静态变量造成的内存泄漏

3、匿名内部类/非静态内部类

4、资源未关闭造成的内存泄漏

1、集合类泄漏

集合类添加元素后，仍引用着集合元素对象，导致该集合中的元素对象无法被回收，从而导致内存泄露。

举个栗子：

```
static List<Object> mList = new ArrayList<>();

    for (int i = 0; i < 100; i++) {

          Object obj = new Object();

        mList.add(obj);

          obj = null;

    }
```

当mList没用的时候，我们如果不做处理的话，这就是典型的占着茅坑不拉屎，mList内部持有者众多集合元素的对象，不泄露天理难容啊。解决这个问题也超级简单。把mList清理掉，然后把它的引用也给释放掉。

```
mList.clear();

mList = null;
```

2、单例/静态变量造成的内存泄漏

单例模式具有其 静态特性，它的生命周期 等于应用程序的生命周期，正是因为这一点，往往很容易造成内存泄漏。

先来一个小栗子：

```
public class SingleInstance {

    private static SingleInstance mInstance;

    private Context mContext;

    private SingleInstance(Context context){

        this.mContext = context;

    }

    public static SingleInstance newInstance(Context context){

        if(mInstance == null){

            mInstance = new SingleInstance(context);

        }

        return sInstance;

    }

}
```

当我们在Activity里面使用这个的时候，把我们Acitivty的context传进去，那么，这个单例就持有这个Activity的引用，当这个Activity没有用了，需要销毁的时候，因为这个单例还持有Activity的引用，所以无法GC回收，所以就出现了内存泄漏，也就是生命周期长的持有了生命周期短的引用，造成了内存泄漏。

所以我们要做的就是生命周期长的和生命周期长的玩，短的和短的玩。就好比你去商场，本来就是传个话的，话说完就要走了，突然保安过来非要拉着你的手，说要和你天长地久。只要商场在一天，他就要陪你一天。天呢？太可怕了。叔叔我们不约，我有我的小伙伴，我还要上学呢，你赶紧找你的保洁阿姨去吧。你在商场的生命周期本来可能就是1分钟，而保安的生命周期那是要和商场开关门一致的，所以不同生命周期的最好别一起玩的好。

解决方案也很简单：

```
public class SingleInstance {

    private static SingleInstance mInstance;

    private Context mContext;

    private SingleInstance(Context context){

        this.mContext = context.getApplicationContext();

    }

    public static SingleInstance newInstance(Context context){

        if(mInstance == null){

            mInstance = new SingleInstance(context);

        }

        return sInstance;

    }

}
```

还有一个常用的地方就是Toast。你应该知道和谁玩了吧。

3、匿名内部类/非静态内部类

这里有一张宝图：

class 对比	static inner class	non static inner cla
与外部 class 引用关系	如果没有传入参数，就没有引用关系	自动获得强引用
被调用时需要外部实例	不需要	需要
能否调用外部 class 中的变量和方法	不能	能
生命周期	自主的生命周期	依赖于外部类，甚至比部类更长

image.png

非静态内部类他会持有他外部类的引用，从图我们可以看到非静态内部类的生命周期可能比外部类更长，这就是二楼的情况一致了，如果非静态内部类的周明周期长于外部类，在加上自动持有外部类的强引用，我的乖乖，想不泄漏都难啊。

我们再来举个栗子：

```
public class TestActivity extends Activity {

    @Override

    protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_test);

        new MyAscnyTask().execute();

    }

    class MyAscnyTask extends AsyncTask<Void, Integer, String>{

        @Override

        protected String doInBackground(Void... params) {

            try {

                Thread.sleep(100000);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

            return "";

        }

    }
```

```
}
```

我们经常会用这个方法去异步加载，然后更新数据。貌似很平常，我们开始学这个的时候就是这么写的，没发现有问题啊，但是你这么想一想，MyAscnyTask是一个非静态内部类，如果他处理数据的时间很长，极端点我们用sleep 100秒，在这期间Activity可能早就关闭了，本来Activity的内存应该被回收的，但是我们知道非静态内部类会持有外部类的引用，所以Activity也需要陪着非静态内部类MyAscnyTask一起天荒地老。好了，内存泄漏就形成了。

怎么办呢？

既然MyAscnyTask的生命周期可能比较长，那就把它变成静态，和Application玩去吧，这样MyAscnyTask就不会再持有外部类的引用了。两者也相互独立了。

```
public class TestActivity extends Activity {

        @Override

        protected void onCreate(Bundle savedInstanceState) {

                super.onCreate(savedInstanceState);

                setContentView(R.layout.activity_test);

                new MyAscnyTask().execute();

        }

//改了这里 注意一下 static

      static    class MyAscnyTask extends AsyncTask<Void, Integer, String>{

                @Override

                protected String doInBackground(Void... params) {

                        try {

                                Thread.sleep(100000);

                        } catch (InterruptedException e) {

                                e.printStackTrace();

                        }

                        return "";

                }

        }

}
```

说完非静态内部类，我再来看看匿名内部类，这个问题很常见，匿名内部类和非静态内部类有一个共同的地方，就是会只有外部类的强引用，所以这哥俩本质是一样的。但是处理方法有些不一样。但是思路绝对一样。换汤不换药。

举个灰常熟悉的栗子：

```
public class TestActivity extends Activity {
```

```
private TextView mText;
        private Handler mHandler = new Handler(){
                @Override
                public void handleMessage(Message msg) {
                        super.handleMessage(msg);
//do something
mText.setText(" do someThing");
                }
        };
        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_test);
mText = findVIewById(R.id.mText);
                //    匿名线程持有 Activity 的引用，进行耗时操作
                new Thread(new Runnable() {
                        @Override
                        public void run() {
                                try {
                                        Thread.sleep(100000);
                                } catch (InterruptedException e) {
                                        e.printStackTrace();
                                }
                        }
                }).start();

                mHandler. sendEmptyMessageDelayed(0, 100000);
        }
```

想必这两个方法是我们经常用的吧，很熟悉，也是这么学的，没感觉不对啊，老师就是这么教的，通过我们上面的分析，还这么想吗？关键是 耗时时间过长，造成内部类的生命周期大于外部类，对弈非静态内部类，我们可以静态化，至于匿名内部类怎么办呢？一样把它变成静态内部类，也就是说尽量不要用匿名内部类。

完事了吗？很多人不注意这么一件事，如果我们在handleMessage方法里进行UI的更新，这个Handler静态化了和Activity没啥关系了，但是比如这个mText，怎么说？全写是activity.mText，看到了吧，持有了Activity的引用，也就是说Handler费劲心思变成静态类，自认为不持有Activity的引用了，准确的说是不自动持有Activity的引用了，但是我们要做UI更新的时候势必会持有Activity的引用，静态类持有非静态类的引用，我们发现怎么又开始内存泄漏了呢？处处是坑啊，怎么办呢？我们这里就要引出弱引用的概念了。

引用分为强引用，软引用，弱引用，虚引用，强度一次递减。

强引用

我们平时不做特殊处理的一般都是强引用，如果一个对象具有强引用，GC宁可OOM也绝不会回收它。看出多强硬了吧。

软引用(SoftReference)

如果内存空间足够，GC就不会回收它，如果内存空间不足了，就会回收这些对象的内存。

弱引用（WeakReference）

弱引用要比软引用,更弱一个级别，内存不够要回收他，GC的时候不管内存够不够也要回收他，简直是弱的一匹。不过GC是一个优先级很低的线程，也不是太频繁进行，所以弱引用的生活还过得去，没那么提心吊胆。

虚引用

用的甚少，我没有用过，如果想了解的朋友，可以自行谷歌百度。

所以我们用弱引用来修饰Activity，这样GC的时候，该回收的也就回收了，不会再有内存泄漏了。很完美。

```
public class TestActivity extends Activity {

    private TextView mText;

    private MyHandler myHandler = new MyHandler(TestActivity.this);

    private MyThread myThread = new MyThread();

    private static class MyHandler extends Handler {

        WeakReference<TestActivity> weakReference;

        MyHandler(TestActivity testActivity) {

            this.weakReference = new WeakReference<TestActivity>(testActivity);

        }

        @Override

        public void handleMessage(Message msg) {

            super.handleMessage(msg);

            weakReference.get().mText.setText("do someThing");

        }

    }

    private static class MyThread extends Thread {
```

```
            @Override
            public void run() {
                super.run();
                try {
                    sleep(100000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_test);
            mText = findViewById(R.id.mText);
            myHandler.sendEmptyMessageDelayed(0, 100000);
            myThread.start();
        }
//最后清空这些回调
        @Override
        protected void onDestroy() {
            super.onDestroy();
            myHandler.removeCallbacksAndMessages(null);
        }
```

4、资源未关闭造成的内存泄漏

- 网络、文件等流忘记关闭
- 手动注册广播时，退出时忘记 unregisterReceiver()
- Service 执行完后忘记 stopSelf()
- EventBus 等观察者模式的框架忘记手动解除注册

这些需要记住又开就有关，具体做法也很简单就不一一赘述了。给大家介绍几个很好用的工具：

1、leakcanary傻瓜式操作，哪里有泄漏自动给你显示出来，很直接很暴力。

2、我们平时也要多使用Memory Monitor进行内存监控，这个分析就有些难度了，可以上网搜一下具体怎么使用。

3、Android Lint 它可以帮助我们发现代码机构 / 质量问题，同时提供一些解决方案，内存泄露的会飘黄，用起来很方便，具体使用方法上网学习，这里不多做说明了。

so

优化3.0

1. 解决各个情况下的内存泄漏，注意平时代码的规范。

4、启动速度优化

不知道大家有没有细心发现，我们的应用启动要比别的大厂的要慢，要花费更多的时间，明明他们的包体更大，app更复杂，怎么启动时间反而比我们的短呢？

但是这块的优化关注的人很少，因为App常常伴有闪屏页，所以这个问题看起来就不是问题了，但是一款好的应用是绝对不允许这样的，我加闪屏页是我的事，启动速度慢绝对不可以。

app启动分为冷启动（Cold start）、热启动（Hot start）和温启动（Warm start）三种。

冷启动（Cold start）

冷启动是指应用程序从头开始：系统的进程在此开始之前没有创建应用程序。冷启动发生在诸如自设备启动以来首次启动应用程序或自系统终止应用程序以来。

在冷启动开始时，系统有三个任务。这些任务是：

1、加载并启动应用程序

2、启动后立即显示应用程序的空白启动窗口

3、创建应用程序进程

当系统为我们创建了应用进程之后，开始创建应用程序对象。

1、启动主线程

2、创建主Activity

3、加载布局

4、屏幕布局

5、执行初始绘制

应用程序进程完成第一次绘制后，系统进程会交换当前显示的背景窗口，将其替换为主活动。此时，用户可以开始使用该应用程序。至此启动完成。

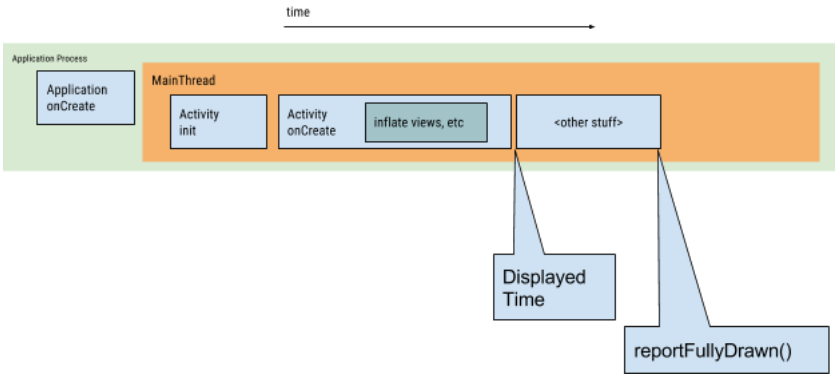


image.png

Application创建

当Application启动时，空白的启动窗口将保留在屏幕上，直到系统首次完成绘制应用程序。此时，系统进程会交换应用程序的启动窗口，允许用户开始与应用程序进行交互。这就是为什么我们的程序启动时会先出现一段时间的黑屏(白屏)。

如果我们有自己的Application，系统会onCreate()在我们的Application对象上调用该方法。之后，应用程序会生成主线程（也称为UI线程），并通过创建主要活动来执行任务。

从这一点开始，App就按照他的 应用程序生命周期阶段进行。

Activity创建

应用程序进程创建活动后，活动将执行以下操作：

1. 初始化值。
2. 调用构造函数。
3. 调用回调方法，例如 Activity.onCreate()，对应Activity的当前生命周期状态。

通常，该 onCreate()方法对加载时间的影响最大，因为它以最高的开销执行工作：加载和膨胀视图，以及初始化活动运行所需的对象。

热启动（Hot start）

应用程序的热启动比冷启动要简单得多，开销也更低。在一个热启动中，系统都会把你的Activity带到前台。如果应用程序的Activity仍然驻留在内存中，那么应用程序可以避免重复对象初始化、布局加载和渲染。

热启动显示与冷启动方案相同的屏幕行为：系统进程显示空白屏幕，直到应用程序完成呈现活动。

温启动（Warm start）

温启动包含了冷启动时发生的一些操作，与此同时，它表示的开销比热启动少，有许多潜在的状态可以被认为是温暖的开始。

场景：

- 用户退出您的应用，但随后重新启动它。该过程可能已继续运行，但应用程序必须通过调用从头开始重新创建Activity的onCreate()。
- 系统将您的应用程序从内存中逐出，然后用户重新启动它。需要重新启动进程和活动，但是在调用onCreate()的时候可以从Bundle（savedInstanceState）获取数据。

了解完启动过程，我们就知道哪里会影响我们启动的速度了。在创建应用程序和创建Activity期间都可能会出现性能问题。

这里是慢的定义：

- 冷启动需要5秒或更长时间。
- 温启动需要2秒或更长时间。

- 热启动需要1.5秒或更长时间。

无论何种启动，我们的优化点都是：

Application、Activity创建以及回调等过程

谷歌官方给的建议是：

1、利用提前展示出来的Window，快速展示出来一个界面，给用户快速反馈的体验；

2、避免在启动时做密集沉重的初始化（Heavy app initialization）；

3、避免I/O操作、反序列化、网络操作、布局嵌套等。

具体做法：

针对1：利用提前展示出来的Window，快速展示出来一个界面

使用Activity的windowBackground主题属性来为启动的Activity提供一个简单的drawable。

Layout XML file:

```
<layer-list xmlns:android="http://schemas.android.com/apk/res/android" android:opacity="opaque">
    <!-- The background color, preferably the same as your normal theme -->
    <item android:drawable="@android:color/white"/>
    <!-- Your product logo - 144dp color version of your app icon -->
    <item>
        <bitmap
            android:src="@drawable/product_logo_144dp"
            android:gravity="center"/>
    </item>
</layer-list>
```

Manifest file:

```
<activity ...
android:theme="@style/AppTheme.Launcher" />
```

这样在启动的时候，会先展示一个界面，这个界面就是Manifest中设置的Style，等Activity加载完毕后，再去加载Activity的界面，而在Activity的界面中，我们将主题重新设置为正常的主题，从而产生一种快的感觉。其实就是个障眼法而已，提前让你看到了假的界面。也算是一种不错的方法，但是治标不治本。

针对2：避免在启动时做密集沉重的初始化

我们审视一下我们的MyApplication里面的操作。初始化操作有友盟，百度，bugly，数据库，IM，神策，图片加载库，网络请求库，广告sdk，地图，推送，等等，这么多需要初始化，Application的任务太重了，启动不慢才怪呢。

怎么办呢？这些还都是必要的，不能不去初始化啊，那就只能异步加载了。但是并不是所有的都可以进行异步处理。这里分情况给出一些建议：

1、比如像友盟，bugly这样的业务非必要的可以的异步加载。

2、比如地图，推送等，非第一时间需要的可以在主线程做延时启动。当程序已经启动起来之后，在进行初始化。

3、对于图片，网络请求框架必须在主线程里初始化了。

同时因为我们一般会有闪屏页面，也可以把延时启动的地图，推动的启动在这个时间段里，这样合理安排时间片的使用。极大的提高了启动速度。

针对3：避免I/O操作、反序列化、网络操作、布局嵌套等。

这个不用多说了，大家应该知道如何去做了，有些上文也有说明。

so

优化4.0

1. 利用提前展示出来的Window，快速展示出来一个界面，给用户快速反馈的体验；
2. 避免在启动时做密集沉重的初始化（Heavy app initialization）；
3. 避免I/O操作、反序列化、网络操作、布局嵌套等。

5、包体优化

我做过两年的海外应用产品，深知包体大小对于产品新增的影响，包体小百分之五，可能新增就增加百分之五。如果产品基数很大，这个提升就更可怕了。不管怎么说，我们要减肥，要六块腹肌，不要九九归一的大肚子。

既然要瘦身，那么我们必须知道APK的文件构成，解压apk：

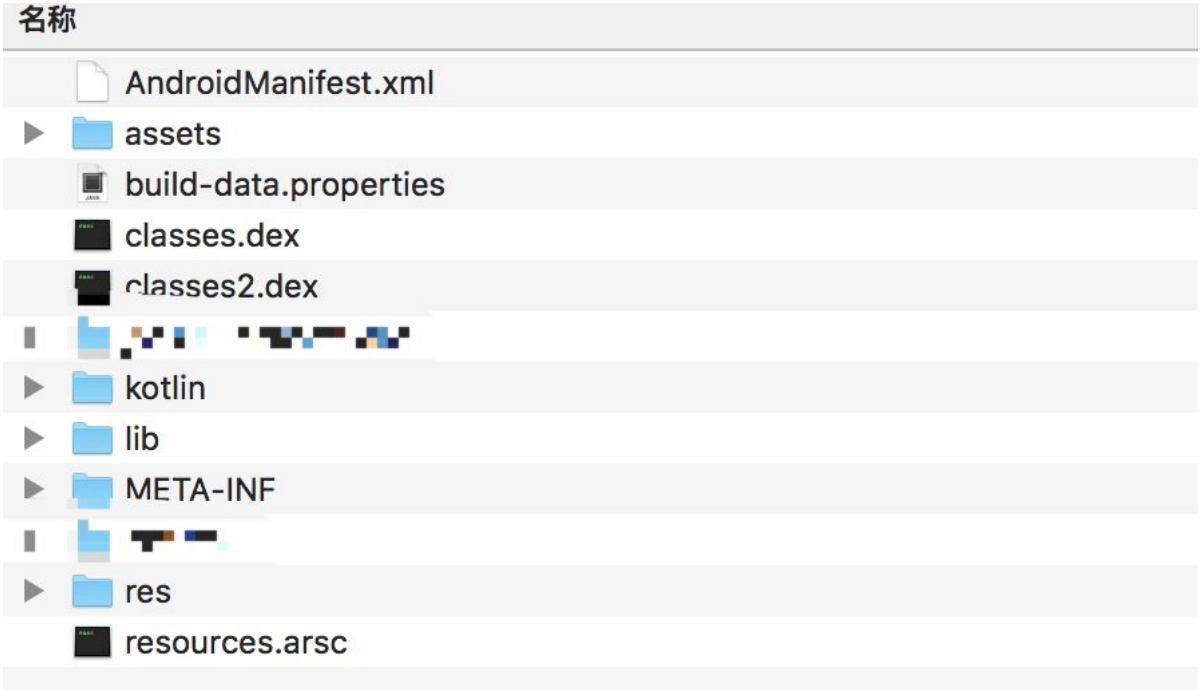


image.png

assets文件夹

存放一些配置文件、资源文件，assets不会自动生成对应的 ID，而是通过 AssetManager 类的接口获取。

res目录

res 是 resource 的缩写，这个目录存放资源文件，会自动生成对应的 ID 并映射到 .R 文件中，访问直接使用资源 ID。

META-INF

保存应用的签名信息，签名信息可以验证 APK 文件的完整性。

AndroidManifest.xml

这个文件用来描述 Android 应用的配置信息，一些组件的注册信息、可使用权限等。

classes.dex

Dalvik 字节码程序，让 Dalvik 虚拟机可执行，一般情况下，Android 应用在打包时通过 Android SDK 中的 dx 工具将 Java 字节码转换为 Dalvik 字节码。

resources.arsc

记录着资源文件和资源 ID 之间的映射关系，用来根据资源 ID 寻找资源。

我们需要从代码和资源两个方面去减少响应的大小。

1、首先我们可以使用lint工具，如果有没有使用过的资源就会打印如下的信息(不会使用的朋友可以上网看一下)

```
res/layout/preferences.xml: Warning: The resource R.layout.preferences appears
        to be unused [UnusedResources]
```

同时我们可以开启资源压缩,自动删除无用的资源

```
android {
        ...
        buildTypes {
                release {
                        shrinkResources true
                        minifyEnabled true
                        proguardFiles getDefaultProguardFile('proguard-android.txt'),
                                      'proguard-rules.pro'
                }
        }
```

无用的资源已经被删除了，接下来哪里可以在瘦身呢？

2、我们可以使用可绘制对象，某些图像不需要静态图像资源；框架可以在运行时动态绘制图像。Drawable对象（<shape>以XML格式）可以占用APK中的少量空间。此外，XML Drawable对象产生符合材料设计准则的单色图像。

上面的话官方，简单说来就是，能自己用XML写Drawable，就自己写，能不用公司的UI切图，就别和他们说话，咱们自己造，做自己的UI，美滋滋。而且这种图片占用空间会很小。

3、重用资源，比如一个三角按钮，点击前三角朝上代表收起的意思，点击后三角朝下，代表展开，一般情况下，我们会用两张图来切换，我们完全可以用旋转的形式去改变

```
<?xml  version="1.0"  encoding="utf-8"?>

<rotate  xmlns:android="http://schemas.android.com/apk/res/android"

        android:drawable="@drawable/ic_thumb_up"

        android:pivotX="50%"

        android:pivotY="50%"

        android:fromDegrees="180"  />
```

比如同一图像的着色不同，我们可以用android:tint和tintMode属性，低版本（5.0以下）可以使用ColorFilter。

4、压缩PNG和JPEG文件

您可以减少PNG文件的大小，而不会丢失使用工具如图像质量 pngcrush，pngquant，或zopflipng。所有这些工具都可以减少PNG文件的大小，同时保持感知的图像质量。

5、使用WebP文件格式

可以使用图像的WebP文件格式，而不是使用PNG或JPEG文件。WebP格式提供有损压缩（如JPEG）以及透明度（如PNG），但可以提供比JPEG或PNG更好的压缩。

可以使用Android Studio将现有的BMP，JPG，PNG或静态GIF图像转换为WebP格式。

6、使用矢量图形

可以使用矢量图形来创建与分辨率无关的图标和其他可伸缩Image。使用这些图形可以大大减少APK大小。一个100字节的文件可以生成与屏幕大小相关的清晰图像。

但是，系统渲染每个VectorDrawable对象需要花费大量时间 ，而较大的图像需要更长的时间才能显示在屏幕上。因此，请考虑仅在显示小图像时使用这些矢量图形。

不要把AnimationDrawable用于创建逐帧动画，因为这样做需要为动画的每个帧包含一个单独的位图文件，这会大大增加APK的大小。

7、代码混淆

使用proGuard 代码混淆器工具，它包括压缩、优化、混淆等功能。这个大家太熟悉了。不多说了。

```
android  {

        buildTypes  {

                release  {

                        minifyEnabled  true
```

```
            proguardFiles  getDefaultProguardFile('proguard-android.txt'),
                           'proguard-rules.pro'
        }
    }
```

8、插件化。

比如功能模块放在服务器上，按需下载，可以减少安装包大小。

so

优化5.0

1. 代码混淆
2. 插件化
3. 资源优化

6、耗电优化

我们可能对耗电优化不怎么感冒，没事，谷歌这方面做得也不咋地，5.0之后才有像样的方案，讲实话这个优化的优先级没有前面几个那么高，但是我们也要了解一些避免耗电的坑，至于更细的耗电分析可以使用这个Battery Historian。

Battery Historian 是由Google提供的Android系统电量分析工具，从手机中导出bugreport文件上传至页面，在网页中生成详细的图表数据来展示手机上各模块电量消耗过程，最后通过App数据的分析制定出相关的电量优化的方法。

我们来谈一下怎么规避电老虎吧。

谷歌推荐使用JobScheduler，来调整任务优先级等策略来达到降低损耗的目的。JobScheduler可以避免频繁的唤醒硬件模块，造成不必要的电量消耗。避免在不合适的时间(例如低电量情况下、弱网络或者移动网络情况下的)执行过多的任务消耗电量。

具体功能：

1、可以推迟的非面向用户的任务(如定期数据库数据更新)；

2、当充电时才希望执行的工作(如备份数据)；

3、需要访问网络或 Wi-Fi 连接的任务(如向服务器拉取配置数据)；

4、零散任务合并到一个批次去定期运行；

5、当设备空闲时启动某些任务；

6、只有当条件得到满足，系统才会启动计划中的任务（充电、WIFI…）。

同时谷歌针对耗电优化也提出了一个懒惰第一的法则：

减少

你的应用程序可以删除冗余操作吗？例如，它是否可以缓存下载的数据而不是重复唤醒无线电以重新下载数据？

推迟

应用是否需要立即执行操作？例如，它可以等到设备充电才能将数据备份到云端吗？

合并

可以批处理工作，而不是多次将设备置于活动状态吗？例如，几十个应用程序是否真的有必要在不同时间打开收音机发送邮件？在一次唤醒收音机期间，是否可以传输消息？

谷歌在耗电优化这方面确实显得有些无力，希望以后可以退出更好的工具和解决方案，不然这方面的优化优先级还是很低。付出和回报所差太大。

so

优化6.0

1. 使用JobScheduler调度任务
2. 使用懒惰法则

6、ListView和 Bitmap优化

针对ListView优化，主要是合理使用ViewHolder。创建一个内部类ViewHolder，里面的成员变量和view中所包含的组件个数、类型相同，在convertview为null的时候，把findviewbyId找到的控件赋给ViewHolder中对应的变量，就相当于先把它们装进一个容器，下次要用的时候，直接从容器中获取。

现在我们现在一般使用RecyclerView，自带这个优化，不过还是要理解一下原理的好。

然后可以对接受来的数据进行分段或者分页加载，也可以优化性能。

对于Bitmap，这个我们使用的就比较多了，很容易出现OOM的问题，图片内存的问题可以看一下我之前写的这篇文章一张图片占用多少内存。

Bitmap的优化套路很简单，粗暴，就是让压缩。

三种压缩方式：

1.对图片质量进行压缩

2.对图片尺寸进行压缩

3.使用libjpeg.so库进行压缩

对图片质量进行压缩

```
public static Bitmap compressImage(Bitmap bitmap){

        ByteArrayOutputStream baos = new ByteArrayOutputStream();

        //质量压缩方法，这里100表示不压缩，把压缩后的数据存放到baos中

        bitmap.compress(Bitmap.CompressFormat.JPEG, 100, baos);

        int options = 100;

        //循环判断如果压缩后图片是否大于50kb,大于继续压缩
```

```
            while ( baos.toByteArray().length / 1024>50) {
                //清空baos
                baos.reset();
                bitmap.compress(Bitmap.CompressFormat.JPEG, options, baos);
                options -= 10;//每次都减少10
            }
            //把压缩后的数据baos存放到ByteArrayInputStream中
            ByteArrayInputStream isBm = new ByteArrayInputStream(baos.toByteArray());
            //把ByteArrayInputStream数据生成图片
            Bitmap newBitmap = BitmapFactory.decodeStream(isBm, null, null);
            return newBitmap;
        }
```

对图片尺寸进行压缩

```
    /**
     * 按图片尺寸压缩 参数是bitmap
     * @param bitmap
     * @param pixelW
     * @param pixelH
     * @return
     */
    public static Bitmap compressImageFromBitmap(Bitmap bitmap, int pixelW, int pixelH) {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        bitmap.compress(Bitmap.CompressFormat.JPEG, 100, os);
        if( os.toByteArray().length / 1024>512) {//判断如果图片大于0.5M,进行压缩避免在生成图片（BitmapFactory.decodeStream）时溢出
            os.reset();
            bitmap.compress(Bitmap.CompressFormat.JPEG, 50, os);//这里压缩50%，把压缩后的数据存放到baos中
        }
        ByteArrayInputStream is = new ByteArrayInputStream(os.toByteArray());
        BitmapFactory.Options options = new BitmapFactory.Options();
        options.inJustDecodeBounds = true;
        options.inPreferredConfig = Bitmap.Config.RGB_565;
```

```
        BitmapFactory.decodeStream(is, null, options);
        options.inJustDecodeBounds = false;
        options.inSampleSize = computeSampleSize(options , pixelH > pixelW ? pixelW : pixelH ,pixelW * pixelH );
        is = new ByteArrayInputStream(os.toByteArray());
        Bitmap newBitmap = BitmapFactory.decodeStream(is, null, options);
        return newBitmap;
}

/**
 * 动态计算出图片的inSampleSize
 * @param options
 * @param minSideLength
 * @param maxNumOfPixels
 * @return
 */
public static int computeSampleSize(BitmapFactory.Options options, int minSideLength, int maxNumOfPixels) {
        int initialSize = computeInitialSampleSize(options, minSideLength, maxNumOfPixels);
        int roundedSize;
        if (initialSize <= 8) {
                roundedSize = 1;
                while (roundedSize < initialSize) {
                        roundedSize <<= 1;
                }
        } else {
                roundedSize = (initialSize + 7) / 8 * 8;
        }
        return roundedSize;
}
private static int computeInitialSampleSize(BitmapFactory.Options options, int minSideLength, int maxNumOfPixels) {
        double w = options.outWidth;
        double h = options.outHeight;
```

```
        int lowerBound = (maxNumOfPixels == -1) ? 1 : (int) Math.ceil(Math.sqrt(w * h / maxNumOfPixels));
        int upperBound = (minSideLength == -1) ? 128 :(int) Math.min(Math.floor(w / minSideLength), Math.floor(h / minSideLength));
        if (upperBound < lowerBound) {
            return lowerBound;
        }
        if ((maxNumOfPixels == -1) && (minSideLength == -1)) {
            return 1;
        } else if (minSideLength == -1) {
            return lowerBound;
        } else {
            return upperBound;
        }
    }
```

使用libjpeg.so库进行压缩

可以参考这篇Android性能优化系列之Bitmap图片优化：

https://blog.csdn.net/u012124438/article/details/66087785)

优化7.0

1. ListView使用ViewHolder，分段，分页加载
2. 压缩Bitmap

8、响应速度优化

影响响应速度的主要因素是主线程有耗时操作，影响了响应速度。所以响应速度优化的核心思想是避免在主线程中做耗时操作，把耗时操作异步处理。

9、线程优化

线程优化的思想是采用线程池，避免在程序中存在大量的Thread。线程池可以重用内部的线程，从而避免了现场的创建和销毁所带来的性能开销，同时线程池还能有效地控制线程池的最大并发数，避免大量的线程因互相抢占系统资源从而导致阻塞现象发生。

《Android开发艺术探索》对线程池的讲解很详细，不熟悉线程池的可以去了解一下。

- **优点：**

1、减少在创建和销毁线程上所花的时间以及系统资源的开销。

2、如不使用线程池，有可能造成系统创建大量线程而导致消耗完系统内存以及”过度切换”。

- **需要注意的是：**

1、如果线程池中的数量为达到核心线程的数量，则直接会启动一个核心线程来执行任务。

2、如果线程池中的数量已经达到或超过核心线程的数量，则任务会被插入到任务队列中标等待执行。

3、如果(2)中的任务无法插入到任务队列中，由于任务队列已满，这时候如果线程数量未达到线程池规定最大值，则会启动一个非核心线程来执行任务。

4、如果(3)中线程数量已经达到线程池最大值，则会拒绝执行此任务，线程池会调用RejectedExecutionHandler的rejectedExecution方法通知调用者。

10、微优化

这些微优化可以在组合时提高整体应用程序性能，但这些更改不太可能导致显着的性能影响。选择正确的算法和数据结构应始终是我们的首要任务，以提高代码效率。

- **编写高效代码有两个基本规则：**

1、不要做你不需要做的工作

2、如果可以避免，请不要分配内存

1、避免创建不必要的对象

对象创建永远不是免费的，虽然每一个的代价不是很大，但是总归是代价的不是吗？能不创建何必要浪费资源呢？

2、首选静态（这里说的是特定情景）

如果您不需要访问对象的字段，请使您的方法保持静态。调用速度将提高约15％-20％。这也是很好的做法，因为你可以从方法签名中看出，调用方法不能改变对象的状态

3、对常量使用static final

此优化仅适用于基本类型和String常量，而不适用于 任意引用类型。尽管如此，static final尽可能声明常量是一种好习惯。

4、使用增强的for循环语法

增强for循环（for-each）可用于实现Iterable接口和数组的集合。对于集合，分配一个迭代器来对hasNext()和进行接口调用next()。使用一个 ArrayList，手写计数循环快约3倍，但对于其他集合，增强的for循环语法将完全等效于显式迭代器用法。

5、避免使用浮点数

根据经验，浮点数比Android设备上的整数慢约2倍

结尾

本文篇幅有限，性能优化的方面很多，每一项深入下去，不写个几十万字是结束不了，所以很多都是浅尝辄止，希望可以抛砖引玉，用我的拙劣的文章，给大家一些帮助。性能优化需要走的路还很远，希望能和各位同学一同前行，一起进步。

参考：

Android APP 性能优化的一些思考

https://www.cnblogs.com/cr330326/p/8011523.html

谷歌官方

http://developer.android.com/topic/performance/

Android性能优化系列之Bitmap图片优化

https://blog.csdn.net/u012124438/article/details/66087785

Android 内存泄漏总结

https://yq.aliyun.com/articles/3009

— — — END — — —

近期文章回顾

- 我对移动端架构的思考
- HTTP 必知必会的那些
- Android 官方架构组件 Paging：分页库的设计美学