

## Android年薪30万面试宝典-不定期更新



作者 小楠总 (/u/70c12759d4fe) +关注

2016.09.03 15:24\* 字数 4771 阅读 1264 评论 2 喜欢 22

(/u/70c12759d4fe)

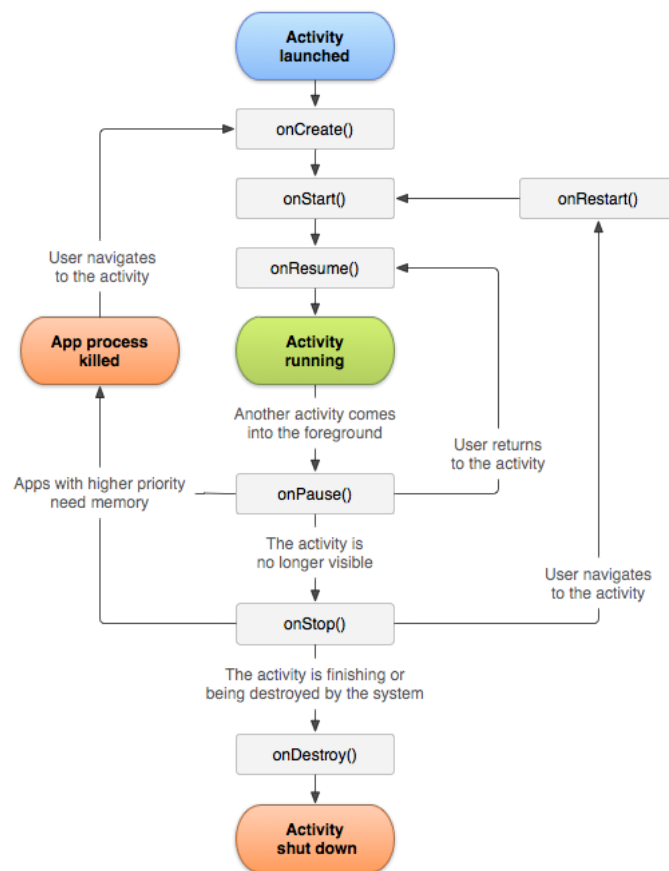
作者-焕然一璐，支持原创，转载请注明出处，谢谢合作。

原文链接：<http://www.jianshu.com/p/4bcd4c50fd6b>

(<http://www.jianshu.com/p/4bcd4c50fd6b>)

# Android年薪30万面试宝典

## 1、Activity的生命周期



生命周期：对象什么时候生，什么时候死，怎么写代码，代码往那里写。

注意：

1. 当打开新的Activity，采用透明主题的时候，当前Activity不会回调onStop
2. onCreate和onDestroy配对，onStart和onStop配对（是否可见），onResume和onPause配对（是否在前台，可以与用户交互）
3. 打开新的Activity的时候，相关的Log为：

```
Main1Activity: onPause
Main2Activity: onCreate
Main2Activity: onStart
Main2Activity: onResume
Main1Activity: onStop
```

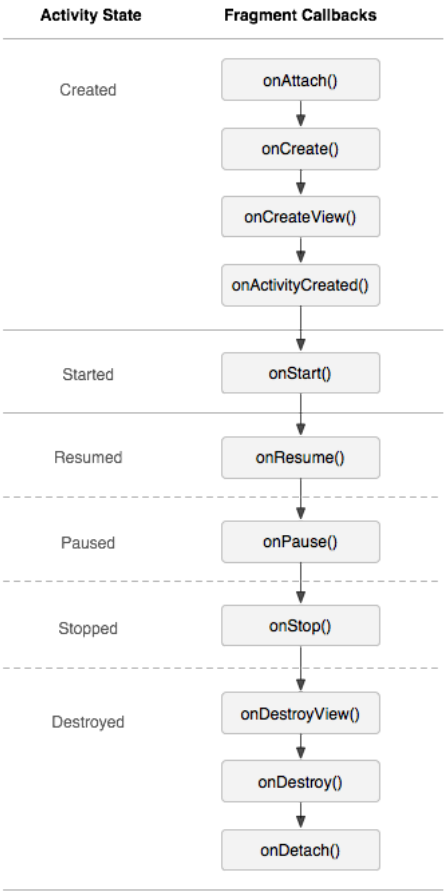


异常状态下的生命周期：

资源相关的系统配置发生改变或者资源不足：例如屏幕旋转，当前Activity会销毁，并且在onStop之前回调onSaveInstanceState保存数据，在重新创建Activity的时候在onStart之后回调onRestoreInstanceState。其中Bundle数据会传到onCreate（不一定有数据）和onRestoreInstanceState（一定有数据）。

防止屏幕旋转的时候重建，在清单文件中添加配置：  
android:configChanges="orientation"

2、Fragment的生命周期



正常启动

Activity: onCreate  
Fragment: onAttach  
Fragment: onCreate  
Fragment: onCreateView  
Fragment: onActivityCreated  
Activity: onStart  
Activity: onResume

正常退出

Activity: onPause  
Activity: onStop  
Fragment: onDestroyView  
Fragment: onDestroy  
Fragment: onDetach  
Activity: onDestroy

3、Activity的启动模式

1. standard：每次激活Activity时(startActivity)，都创建Activity实例，并放入任务栈；



2. singleTop：如果某个Activity自己激活自己，即任务栈栈顶就是该Activity，则不需要创建，其余情况都要创建Activity实例；
3. singleTask：如果要激活的那个Activity在任务栈中存在该实例，则不需要创建，只需要把此Activity放入栈顶，即把该Activity以上的Activity实例都pop，并调用其onNewIntent；
4. singleInstance：应用1的任务栈中创建了MainActivity实例，如果应用2也要激活MainActivity，则不需要创建，两应用共享该Activity实例。

## 4、Activity与Fragment之间的传值

1. 通过findFragmentByTag或者getActivity获得对方的引用（强转）之后，再相互调用对方的public方法，但是这样做一是引入了“强转”的丑陋代码，另外两个类之间各自持有对方的强引用，耦合较大，容易造成内存泄漏。
2. 通过Bundle的方法进行传值，例如以下代码：

```
//Activity中对fragment设置一些参数
fragment.setArguments(bundle);

//fragment中通过getArguments获得Activity中的方法
Bundle arguments = getArguments();
```

3. 利用eventbus进行通信，这种方法实时性高，而且Activity与Fragment之间可以完全解耦。

```
//Activity中的代码
EventBus.getDefault().post("消息");

//Fragment中的代码
EventBus.getDefault().register(this);

@Subscribe
public void test(String text) {
    tv_test.setText(text);
}
```

## 5、Service

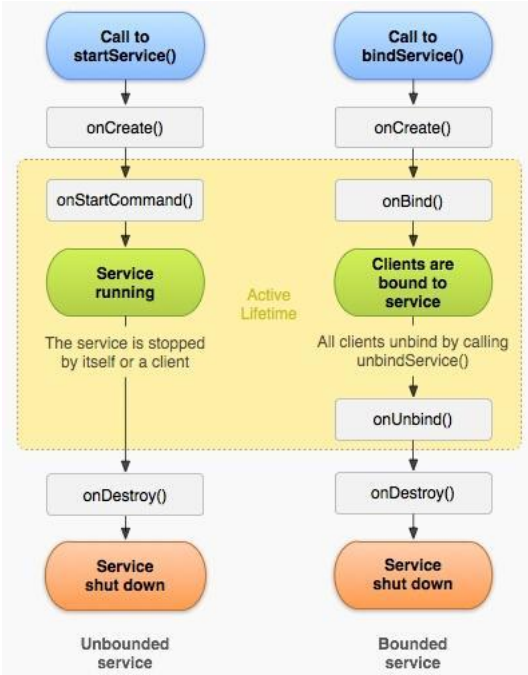
Service分为两种：

1. 本地服务，属于同一个应用程序，通过startService来启动或者通过bindService来绑定并且获取代理对象。如果只是想开个服务在后台运行的话，直接startService即可，如果需要相互之间进行传值或者操作的话，就应该通过bindService。
2. 远程服务（不同应用程序之间），通过bindService来绑定并且获取代理对象。

对应的生命周期如下：

```
context.startService() -> onCreate() -> onStartCommand() -> Service running -- 调用context.stopService()
context.bindService() -> onCreate() -> onBind() -> Service running -- 调用onUnbind() -> onDestroy()
```





注意

Service默认是运行在main线程的，因此Service中如果需要执行耗时操作（大文件的操作，数据库的拷贝，网络请求，文件下载等）的话应该在子线程中完成。

！特殊情况是：Service在清单文件中指定了在其他进程中运行。

6、Android中的消息传递机制

为什么要使用Handler？

因为屏幕的刷新频率是60Hz，大概16毫秒会刷新一次，所以为了保证UI的流畅性，耗时操作需要在子线程中处理，子线程不能直接对UI进行更新操作。因此需要Handler在子线程发消息给主线程来更新UI。

这里再深入一点，Android中的UI控件不是线程安全的，因此在多线程并发访问UI的时候会导致UI控件处于不可预期的状态。Google不通过锁的机制来处理这个问题是因为：

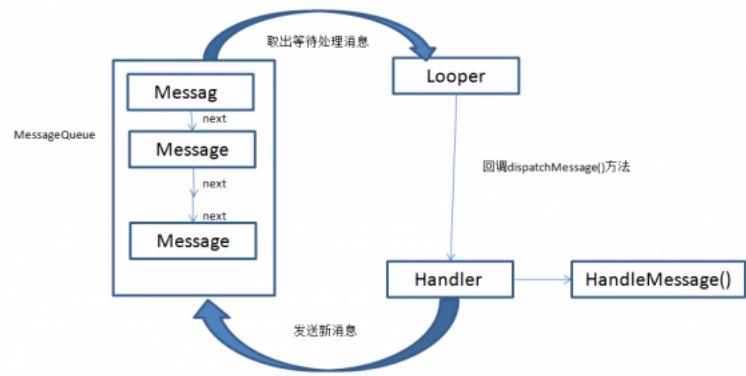
- 1. 引入锁会导致UI的操作变得复杂
- 2. 引入锁会导致UI的运行效率降低

因此，Google的工程师最后是通过单线程的模型来操作UI，开发者只需要通过Handler在不同线程之间切花就可以了。

概述一下Android中的消息机制？

Android中的消息机制主要是指Handler的运行机制。Handler是进行线程切换的关键，在主线程和子线程之间切换只是一种比较特殊的使用情景而已。其中消息传递机制需要了解的东西有Message、Handler、Looper、Looper里面的MessageQueue对象。





如上图所示，我们可以把整个消息机制看作是一条流水线。其中：

- 1. MessageQueue是传送带，负责Message队列的传送与管理
- 2. Looper是流水线的发动机，不断地把消息从消息队列里面取出来，交给Handler来处理
- 3. Message是每一件产品
- 4. Handler就是工人。但是这么比喻不太恰当，因为发送以及最终处理Message的都是Handler

为什么在子线程中创建Handler会抛异常？

Handler的工作是依赖于Looper的，而Looper（与消息队列）又是属于某一个线程（ThreadLocal是线程内部的数据存储类，通过它可以在指定线程中存储数据，其他线程则无法获取到），其他线程不能访问。因此Handler就是间接跟线程是绑定在一起了。因此要使用Handler必须要保证Handler所创建的线程中有Looper对象并且启动循环。因为子线程中默认是没有Looper的，所以会报错。

正确的使用方法是：

```
handler = null;
new Thread(new Runnable() {

    private Looper mLooper;

    @Override
    public void run() {
        //必须调用Looper的prepare方法为当前线程创建一个Looper对象，然后启动循环
        //prepare方法中实质是给ThreadLocal对象创建了一个Looper对象
        //如果当前线程已经创建过Looper对象了，那么会报错
        Looper.prepare();
        handler = new Handler();
        //获取Looper对象
        mLooper = Looper.myLooper();
        //启动消息循环
        Looper.loop();

        //在适当的时候退出Looper的消息循环，防止内存泄漏
        mLooper.quit();
    }
}).start();
```

主线程中默认是创建了Looper并且启动了消息的循环的，因此不会报错：应用程序的入口是ActivityThread的main方法，在这个方法里面会创建Looper，并且执行Looper的loop方法来启动消息的循环，使得应用程序一直运行。

子线程中可以通过Handler发送消息给主线程吗？

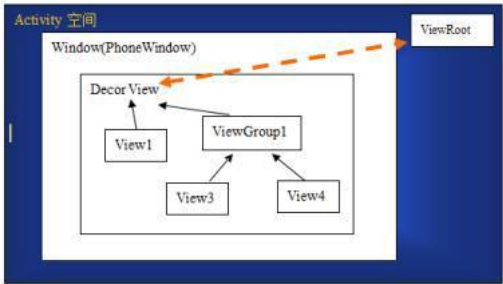
可以。有时候出于业务需要，主线程可以向子线程发送消息。子线程的Handler必须按照上述方法创建，并且关联Looper。



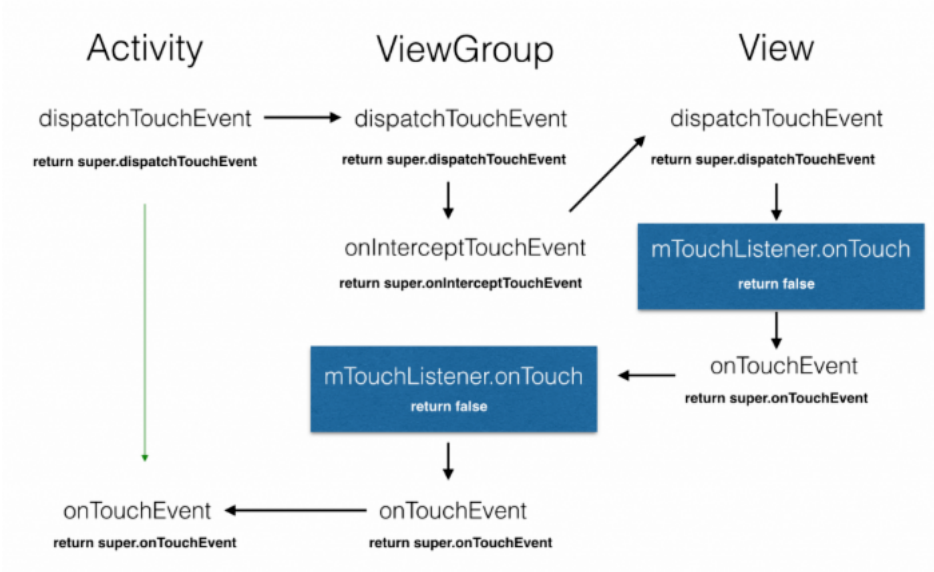
7、事件传递机制以及自定义View相关

Android的视图树

Android中View的机制主要是Activity的显示，每个Activity都有一个Window（具体在手机中的实现类是PhoneWindow），Window以下有DecorView，DecorView下有TitleView以及ContentView，而ContentView就是我们在Activity中通过setContentView指定的。



事件分发机制



ViewGroup有以下三个与事件分发的方法，而View只有dispatchTouchEvent和onTouchEvent。

```
@Override
public boolean dispatchTouchEvent(MotionEvent ev) {
    return super.dispatchTouchEvent(ev);
}

@Override
public boolean onInterceptTouchEvent(MotionEvent ev) {
    return super.onInterceptTouchEvent(ev);
}

@Override
public boolean onTouchEvent(MotionEvent event) {
    return super.onTouchEvent(event);
}
```

事件总是从上往下进行分发，即先到达Activity，再到达ViewGroup，再到达子View，如果没有任何视图消耗事件的话，事件会顺着路径往回传递。其中：

- 1. dispatchTouchEvent是事件的分发方法，如果事件能够到达该视图的话，就首先一定会调用，一般我们不会去修改这个方法。
- 2. onInterceptTouchEvent是事件分发的核心方法，表示ViewGroup是否拦截事件，如果返回true表示拦截，在这之后ViewGroup的onTouchEvent会被调用，事件就不会往下传递。
- 3. onTouchEvent是最低级的，在事件分发中最后被调用。



4. 子View可以通过requestDisallowInterceptTouchEvent方法去请求父元素不要拦截。

### 注意

1. 事件从Activity.dispatchTouchEvent()开始传递，只要没有被停止或拦截，从最上层的View(ViewGroup)开始一直往下(子View)传递。子View 可以通过onTouchEvent()对事件进行处理。
2. 事件由父View(ViewGroup)传递给子View， ViewGroup 可以通过onInterceptTouchEvent()对事件做拦截，停止其往下传递。
3. 如果事件从上往下传递过程中一直没有被停止，且最底层子View 没有消费事件，事件会反向往上传递，这时父View(ViewGroup)可以进行消费，如果还是没有被消费的话，最后会到Activity 的onTouchEvent()函数。
4. 如果View 没有对ACTION\_DOWN 进行消费，之后的其他事件不会传递过来。
5. onTouchListener 优先于onTouchEvent()对事件进行消费。

### 自定义View的分类

1. 对现有的View的子类进行扩展，例如复写onDraw方法、扩展新功能等。
2. 自定义组合控件，把常用一些控件组合起来以方便使用。
3. 直接继承View实现View的完全定制，需要完成View的测量以及绘制。
4. 自定义ViewGroup，需要复写onLayout完成子View位置的确定等工作。

### View的测量-onMeasure

View的测量最终是在onMeasure方法中通过setMeasuredDimension把代表宽高两个MeasureSpec设置给View，因此需要掌握MeasureSpec。MeasureSpec包括大小信息以及模式信息。

MeasureSpec的三种模式：

1. EXACTLY模式：精确模式，对应于用户指定为match\_parent或者具体大小的时候（实际上指定为match\_parent实质上是指定大小为父容器的大小）
2. AT\_MOST模式：对应于用户指定为wrap\_content，此时控件尺寸只要不超过父控件允许的最大尺寸即可。
3. UNSPECIFIED模式：不指定大小的测量模式，这种模式比较少用

下面给出模板代码：



```

public class MeasureUtils {
    /**
     * 用于View的测量
     *
     * @param measureSpec
     * @param defaultSize
     * @return
     */
    public static int measureView(int measureSpec, int defaultSize) {

        int measureSize;

        //获取用户指定的大小以及模式
        int mode = View.MeasureSpec.getMode(measureSpec);
        int size = View.MeasureSpec.getSize(measureSpec);

        //根据模式去返回大小
        if (mode == View.MeasureSpec.EXACTLY) {
            //精确模式 ( 指定大小以及match_parent ) 直接返回指定的大小
            measureSize = size;
        } else {
            //UNSPECIFIED模式、AT_MOST模式 ( wrap_content ) 的话需要提供默认的大小
            measureSize = defaultSize;
            if (mode == View.MeasureSpec.AT_MOST) {
                //AT_MOST ( wrap_content ) 模式下, 需要取测量值与默认值的最小值
                measureSize = Math.min(measureSize, defaultSize);
            }
        }
        return measureSize;
    }
}

```

最后，复写onMeasure方法，把super方法去掉：

```

@Override
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    setMeasuredDimension(MeasureUtils.measureView(widthMeasureSpec, 200),
        MeasureUtils.measureView(heightMeasureSpec, 200)
    );
}

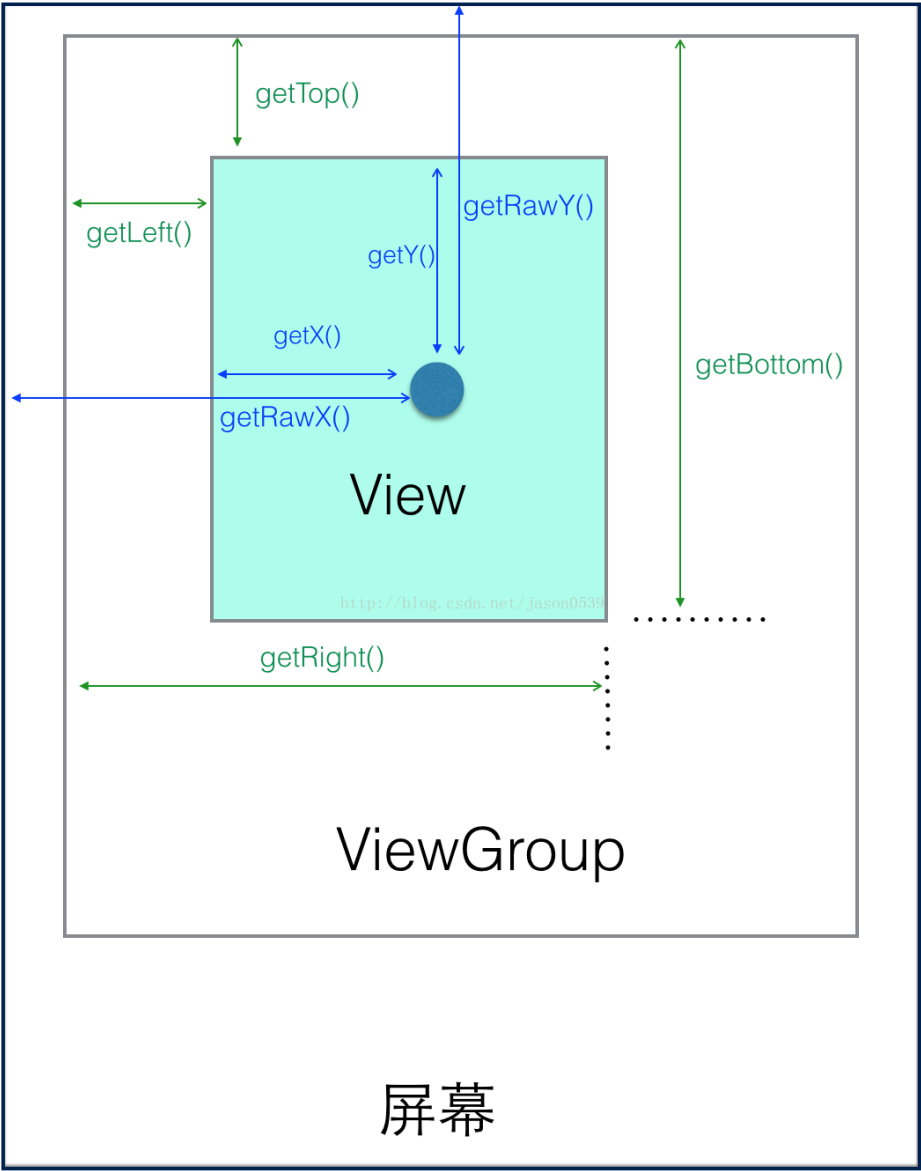
```

## View的绘制-onDraw

View绘制，需要掌握Android中View的坐标体系：







View的坐标体系是以左上角为坐标原点，向右为X轴正方向，向下为Y轴正方向。

View绘制，主要是通过Android的2D绘图机制来完成，时机是onDraw方法中，其中包括画布Canvas，画笔Paint。下面给出示例代码。相关API不是介绍的重点，重点是Canvas的save和restore方法，通过save以后可以对画布进行一些放大缩小旋转倾斜等操作，这两个方法一般配套使用，其中save的调用次数可以多于restore。

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);

    Bitmap bitmap = ImageUtils.drawable2Bitmap(mDrawable);
    canvas.drawBitmap(bitmap, getLeft(), getTop(), mPaint);

    canvas.save();
    //注意，这里的旋转是指画布的旋转
    canvas.rotate(90);
    mPaint.setColor(Color.parseColor("#FF4081"));
    mPaint.setTextSize(30);
    canvas.drawText("测试", 100, -100, mPaint);

    canvas.restore();
}
```

View的位置-onLayout



与布局位置相关的是onLayout方法的复写，一般我们自定义View的时候，只需要完成测量，绘制即可。如果是自定义ViewGroup的话，需要做的就是onLayout中测量自身以及控制子控件的布局位置，onLayout是自定义ViewGroup必须实现的方法。

## 8、性能优化

### 布局优化

1. 使用include标签，通过layout属性复用相同的布局。

```
<include
    android:id="@+id/v_test"
    layout="@layout/include_view" />
```

2. 使用merge标签，去除同类的视图

3. 使用ViewStub来进行布局的延迟加载一些不是马上就用到的布局。例如列表页中，列表在没有拿到数据之前不加载，这样做可以使UI变得流畅。

```
<ViewStub
    android:id="@+id/v_stub"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout="@layout/view_stub" />

//需要手动调用inflate方法，布局才会显示出来。
stub.inflate();
//其中setVisibility在底层也是会调用inflate方法
//stub.setVisibility(View.VISIBLE);
//之后，如果要使用ViewStub标签里面的View，只需要按照平常来即可。
TextView tv_1 = (TextView) findViewById(R.id.tv_1);
```

4. 尽量多使用RelativeLayout，因为这样可以大大减少视图的层级。

### 内存优化

APP设计以及代码编写阶段都应该考虑内存优化：

1. 珍惜Service，尽量使得Service在使用的时候才处于运行状态。尽量使用IntentService

IntentService在内部其实是通过线程以及Handler实现的，当有新的Intent到来的时候，会创建线程并且处理这个Intent，处理完毕以后就自动销毁自身。因此使用IntentService能够节省系统资源。

2. 内存紧张的时候释放资源（例如UI隐藏的时候释放资源等）。复写Activity的回调方法。

```
@Override
public void onLowMemory() {
    super.onLowMemory();
}
```



```
@Override
public void onTrimMemory(int level) {
    super.onTrimMemory(level);

    switch (level) {
        case TRIM_MEMORY_COMPLETE:
            //...
            break;
        case 其他:
    }
}
```

1. 通过Manifest中对Application配置更大的内存，但是一般不推荐

```
android:largeHeap="true"
```

2. 避免Bitmap的浪费，应该尽量去适配屏幕设备。尽量使用成熟的图片加载框架，Picasso，Fresco，Glide等。
3. 使用优化的容器，SparseArray等
4. 其他建议：尽量少用枚举变量，尽量少用抽象，尽量少增加类，避免使用依赖注入框架，谨慎使用library，使用代码混淆，时当场合考虑使用多进程等。
5. 避免内存泄漏（本来应该被回收的对象没有被回收）。一旦APP的内存短时间内快速增长或者GC非常频繁的时候，就应该考虑是否是内存泄漏导致的。

#### 分析方法

1. 使用Android Studio提供的Android Monitors中Memory工具查看内存的使用以及没使用的情况。
2. 使用DDMS提供的Heap工具查看内存使用情况，也可以手动触发GC。
3. 使用性能分析的依赖库，例如Square的LeakCanary，这个库会在内存泄漏的前后通过Notification通知

## 什么情况会导致内存泄漏

1. 资源释放问题：程序代码的问题，长期保持某些资源，如Context、Cursor、IO 流的引用，资源得不到释放造成内存泄露。
2. 对象内存过大问题：保存了多个耗用内存过大的对象（如Bitmap、XML 文件），造成内存超出限制。
3. static 关键字的使用问题：static 是Java 中的一个关键字，当用它来修饰成员变量时，那么该变量就属于该类，而不是该类的实例。所以用static 修饰的变量，它的生命周期是很长的，如果用它来引用一些资源耗费过多的实例（Context 的情况最多），这时就要谨慎对待了。

#### 解决方案

1. 应该尽量避免static 成员变量引用资源耗费过多的实例，比如Context。
2. Context 尽量使用ApplicationContext，因为Application 的Context 的生命周期比较长，引用它不
3. 使用WeakReference 代替强引用。比如可以使用WeakReference<Context> mContextRef

4. 线程导致内存溢出：线程产生内存泄露的主要原因在于线程生命周期的不可控。例如Activity中的Thread在run了，但是Activity由于某种原因重新创建了，但是Thread仍然会运行，因为run方法不结束的话Thread是不会销毁的。

#### 解决方案

1. 将线程的内部类，改为静态内部类（因为非静态内部类拥有外部类对象的强引用，而静态类则不拥有）。
2. 在线程内部采用弱引用保存Context 引用。



## 性能优化

1. 防止过度绘制，通过打开手机的“显示过度绘制区域”即可查看过度绘制的情况。
2. 最小化渲染时间，使用视图树查看节点，对节点进行性能分析。
3. 通过TraceView进行数据的采集以及分析。在有大概定位的时候，使用Android官方提供的Debug类进行采集。最后通过DDMS即可打开这个.trace文件，分析函数的调用情况（包括在指定情况下执行时间，调用次数）

```
//开启数据采集
Debug.startMethodTracing("test.trace");
//关闭
Debug.stopMethodTracing();
```

## OOM

避免OOM的一些常见方法：

1. App资源中尽量少用大图。使用Bitmap的时候要注意等比例缩小图片，并且注意Bitmap的回收。

```
BitmapFactory.Options options = new BitmapFactory.Option();
options.inSampleSize = 2;
//Options 只保存图片尺寸大小，不保存图片到内存
BitmapFactory.Options opts = new BitmapFactory.Options();
opts.inSampleSize = 2;
Bitmap bmp = null;
bmp = BitmapFactory.decodeResource(getResources(),
mImageIds[position],opts);

//回收
bmp.recycle();
```

2. 结合组件的生命周期，释放资源
3. IO流，数据库查询的游标等应该在使用完之后及时关闭。
4. ListView中应该使用ViewHolder模式缓存convertView
5. 页面切换的时候尽量去传递（复用）一些对象

## ANR

不同的组件发生ANR 的时间不一样，主线程（Activity、Service）是5秒，BroadcastReceiver 是10秒。

ANR一般有三种类型：

1. KeyDispatchTimeout(5 seconds)  
主要类型按键或触摸事件在特定时间内无响应
2. BroadcastTimeout(10 seconds)  
BroadcastReceiver在特定时间内无法处理完成
3. ServiceTimeout(20 seconds)  
小概率类型Service在特定的时间内无法处理完成

解决方案：

1. UI线程只进行UI相关的操作。所有耗时操作，比如访问网络，Socket 通信，查询大量SQL 语句，复杂逻辑
2. 无论如何都要确保用户界面操作的流畅度。如果耗时操作需要让用户等待，那么可以在界面上显示进度条。
3. BroadcastReceiver要进行复杂操作的时候，可以在onReceive()方法中启动一个Service来处理。

## 9、九切图（.9图）



点九图，是Android开发中用到的一种特殊格式的图片，文件名以".9.png"结尾。这种图片能告诉程序，图像哪一部分可以被拉升，哪一部分不能被拉升需要保持原有比例。运用点九图可以保证图片在不模糊变形的前提下做到自适应。点九图常用于对话框背景图片中。



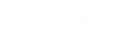
设计稿



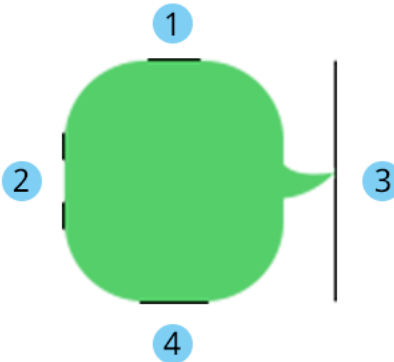
普通切图



点九图



- 1. 1、2部分规定了图像的可拉伸部分,当实际程序中设定了对话框的宽高时，1、2部分就会被拉伸成所需要的高和宽，呈现出于设计稿一样的视觉效果。
- 2. 而3、4部分规定了图像的内容区域。内容区域规定了可编辑区域，例如文字需要被包裹在其内。



9-patch.png

Android中数据常见存储方式

1.





小楠总 (/u/70c12759d4fe)

写了 123928 字，被 437 人关注，获得了 417 个喜欢 (/u/70c12759d4fe)




+ 关注

用最多的梦去面对未来！

如果觉得我的文章对您有用，请随意赞赏。您的支持将鼓励我继续创作！

赞赏支持

喜欢 (/sign\_in) | 22



更多分享

(http://cwb.assets.jianshu.io/notes/images/560391f



登录 (/sign\_in) 后发表评论

2条评论 

只看作者

按喜欢排序 按时间正序 按时间倒序




PeterAtAndroid (/u/4c9330a1ccc7)

2楼 · 2017.02.21 12:16 (/u/4c9330a1ccc7)

thanks.

👍 赞

💬 回复



玉折玲珑曲、 (/u/254cdf12e198)

3楼 · 2017.03.08 09:31 (/u/254cdf12e198)

写的不错，赞👍

👍 赞

💬 回复

被以下专题收入，发现更多相似内容

app开发

Android. ..

Android. ..

面试总结

android 面试

