性能优化

主要内容

· 布局优化

· 绘制优化

· Bitmap优化

· ListView优化

· 内存泄漏优化

布局优化

方法

·删去无用的控件和属性

·减少布局文件的层次
·选择性能较低的ViewGroup
例如，布局中如果LinearLayout可以满足RelativeLayout的
需求，就采用LinearLayout，这是因为RelativeLayout的功能复杂（布局花费时间多）。
但是当LinearLayout需要嵌套实现UI效果，则建议使用RelativeLayout（减少布局文件的层次）

< include >标签

什么是< include >标签

< include >标签就是在一个布局中，引入另一个布局

eg:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent">


    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="这是一个include样例"/>

</LinearLayout>
```

```
<androidx.appcompat.widget.LinearLayoutCompat
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <include android:layout="@layout/title"></include>

</androidx.appcompat.widget.LinearLayoutCompat>
```

为什么使用< include >标签

相同的页面只需要写一次，在需要的地方include即可，提高了共通布局的复用性。

注意事项

·< include >标签只支持以android:layout_开头的属性,比如android:layout_height等，
诸如android:background之类的属性无法使用。但是,android:id这个属性可以使用的，
如果在同一个布局里include引用了多次相同布局，可以修改Id进行区分

< merge >标签

什么是< merge >标签

< merge >标签通常和< include >标签一起使用，以达到减少布局的层级。我们先看上面的栗子，

eg:

```
<androidx.appcompat.widget.LinearLayoutCompat
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <LinearLayout
    android:orientation="vertical" android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="这是一个include样例"/>

    </LinearLayout>

</androidx.appcompat.widget.LinearLayoutCompat>
```

< merge >标签则可以将嵌套时候多余的LinearLayout去掉

ViewStub

什么是ViewStub

ViewStub继承了View，它非常轻量级且宽/高都是0（覆写了onMeasure()）.这是因为它本身是不参与任何的布局
和绘制过程的，也就意味着它可以进行懒加载（延迟加载），对系统占用率就减少惹。

ViewStub的好处

在开发的过程中，我们通常会遇到这样的情况： Json解析出来是空数据、网络异常等。
这个时候就没必要把整个界面初始化的时候就加载进来惹，通过ViewStub就可以在使用的时候再加载，提高程序的初始化性能。

eg:

```
    <ViewStub
        android:id="@+id/stub_import"
        android:inflatedId="@id/title"
        android:layout="@layout/title"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom"/>
```

其中stub_import是ViewStub本身的id，title则是它的根布局的id（也就是上面写过的title布局）
当我们需要启动ViewStub中的根布局的时候可以用以下代码启动

```
var stub:ViewStub=findViewBy(R.id.stub_import)
stub.inflate()
//stub.visibility=View.VISIBLE
```

注意事项

·ViewStub如果重复加载子布局，则会报错.这是因为当子布局加载完成后，子布局会取代ViewStub的位置，并且ViewStub的inflate子布局被置为空
（生完子体，总不能把子体再塞回母体
·ViewStub不能和< merge >标签一起使用
·ViewStub一次只能用Inflate一个布局文件

绘制优化

View的onDraw（）方法避免执行大量的操作

按照Google官方给出的性能优化典范的标准，View的绘制帧率最好保持在60fps
这就要求每帧的绘制时间不超过16ms（1000/60)

方法

·降低绘制复杂度
·onDraw方法中不创建局部对象，因为onDraw()可能被频繁调用，这样就会在一瞬间产生大量的临时对象，这就会占据过多的内存并且导致gc

Bitmap优化

Bitmap的高效加载

Bitmap的加载过程

BitmapFactory类提供了四类方法：decodeFile、decocdeResource、decodeStream和decodeByteArray，分别支持从文件系统、资源、输入流以及字节数组中加载出一个Bitmap对象，其中decodeFile和decodeResource又间接调用了decodeStream方法，这四类方法都是在Android的底层实现的，对应着Bitmap类的几个native方法

核心思想

采用BitmapFactory.Options来加载所需尺寸的图片。这里假设通过ImageView来显示图片，很多时候ImageView并没有图片的原始尺寸那么大，这个时候把整个图片加载进来后设置给ImageView就没有那么必要了，因为ImageView没有办法显示这样原始的图片

所以，我们可以通过BitmapFactory.Options就可以按照一定的采样频率来加载缩小后的推按，将缩小后的图片在ImageView中显示，这样就可以降低内存占用率，提高Bitmap加载时的性能惹

如何获得采样率

1.将BitmapFactory.Options的inJustDecodeBounds参数设为true并加载图片

2.从BitmapFactory.Options中取出图片的原始宽高信息，它们对应于outWidth和outHeight参数

3.根据采样率的规则并结合目标View的所需大小计算出采样率inSampleSize

4.将BitmapFactory.Options的inJustDecodeBounds参数设为false，然后重新加载图片

以从文件系统中快速加载Bitmap为例

```
oncraete(.....){
    .......
var mImageView = findViewById<ImageView>(R.id.iv_test)
        mImageView.setImageBitmap(decodeSampleBitmapFromResource(resources,
R.drawable.ic_bitmap_test, 100, 100))
.....
}

    fun decodeSampleBitmapFromResource(
        res:Resources,resID:Int,reqWidth:Int,reqHeight:Int): Bitmap
{
        //第一步，将inJustDecodeBounds参数设置为true并且加载图片
        var options: BitmapFactory.Options=BitmapFactory.Options();
        options.inJustDecodeBounds=true;
        BitmapFactory.decodeResource(res,resID,options);

        //第二步，计算采样率inSampleSize
        options.inSampleSize=calculateInSampleSize(options,reqWidth,reqHeight);

        //第三步，inJustDecodeBounds设置为false,Bitmap重新加载图片
        options.inJustDecodeBounds=false;
        return BitmapFactory.decodeResource(res,resID,options);
    }

    fun calculateInSampleSize(
        options:BitmapFactory.Options,reqWidth:Int,reqHeight:Int):Int
{
        //取出BitmapFactory.Options中的图片初始宽高
        var height=options.outHeight
        var width=options.outWidth
        var inSampleSize=1

        if(height>reqHeight || width>reqWidth){
            var halfHeight=height/2
            var halfWidth=width/2
            //计算出采样率
            while(halfHeight/inSampleSize>= reqHeight &&
halfWidth/inSampleSize>=reqWidth){
                inSampleSize*=2
            }
        }
        return inSampleSize;
    }
```

ListView优化

不在getView方法中进行耗时操作

不在getView（）方法中进行耗时操作，遇到加载图片等需求时候，采取异步操作，比如ImageLoader（可以去瞅瞅LruCache缓存类的优化机制）

控制异步任务的执行频率

和前面提到的不要在Ondraw方法中创建局部对象是一个道理，如果用户频繁上下滑动（我就喜欢没事不断滑动知乎主页，啥也不点），这就会在一个瞬间产生大量的异步任务，那么线程池不仅会拥堵而且会带来大量的UI更新，且UI更新是在主线程里完成的，那么会造成一定程度的卡顿。

解决方法：修改滑动监听，当滑动结束的时候再加载图片

```
companion object{
    var mIsGridViewIdle:Boolean=false;
}
//scrollState的三种状态
//屏幕惯性滑动时SCROLL_STATE_FLING
//屏幕滚动时，手指还停留在屏幕上SCROLL_STATE_TOUCH_SCROLL
//已经停止SCROLL_STATE_IDLE
public fun onScrollSateChanged(view: AbsListView,scrollState:Int) {
    if(scrollState==AbsListView.OnScrollListener.SCROLL_STATE_IDLE){
        mIsGridViewIdle=true;
        //如果停止了，告诉ListView的adapter数据更新
        mImageAdapter.notifyDataSetChanged;
   }else{
        mIsGridViewIdle=false;
    }
}
-----------------------------------------------------------------------------------
-
//在getView方法中添加这样发发即可
if(mIsGridViewIdle){
    imageView.set.......
    .......
    .......
}
```

硬件加速

如果前面两种方法使用后，滑动依旧有卡顿，可以采用硬件加速,通过设置

```
//androidmenifest.xml文件中对Activity的硬件提速，但是这是一种牺牲内存空间提速的方法，请斟酌使用
android:hardwareAccelerated="true";
```

内存泄漏优化

旧知识回顾

JVM回收对象的经典算法

1.引用计数法

给对象中添加一个引用计数器，每当有一个地方引用他时，计数器值就+1,；当引用失效时，计数器值就-1；任何时刻计数器为0的对象就是不可能在被使用。

可达性分析算法(Reachability Analysis)
2、可达性分析算法（根搜索算法，和DFS深度优先搜索差不多)
通过一系列的GC Roots的对象作为起始点，从这些根节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。

为什么会产生内存泄漏

当一个对象已经不再需要使用的时候，应该被回收，但是另一个正在使用的对象持有它的引用，导致它不能被回收。

听起来挺绕的，借用下上节课ts的奇妙比喻就是.....你跟前女友分手了，却还占着人家不放，不让人家离开，这就造成了内存泄漏

几种内存泄漏的优化

静态变量导致的内存泄漏

```
//我觉得没有人会这么干
//view是一个静态变量，他内部持有了当前的Activity，所以这个Activity无法正常销毁
public class MainActivity extends Activity{
    private static View view;
    .......

     @Override
        protected void onCreate(Bundle savedInstanceState){
        ......
        ......
        view=new View(this);
    }
}
```

单例模式导致的内存泄漏

```
//单例模式：构造函数私有化，提供公共的可访问的获取该对象实例的方法。
public class SingleInstance {
    private Context mContext;
    private static SingleInstance instance;

    private SingleInstance(Context context) {
        this.mContext = context;
    }

    public static SingleInstance getInstance(Context context) {
        if (instance == null) {
            instance = new SingleInstance(context);
        }
        return instance;
    }
}
```

```
-------------------------------------------------------------------------
public class MainActivity extends Activity {
   ........
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        SingleInstance instance
=SingleInstance.getInstance(SecondActivity.this);
    }
```

上面这一串代码中，单例SingleInstance是持有了MainActivity的context。而且很显然，SingleInstance的生命周期比MainActivity长，和上一个例子的静态变量同理。

解决方法

```
public class SingleInstance {
    private Context mContext;
    private static SingleInstance instance;

    private SingleInstance(Context context) {
        this.mContext = context;
    }

    //ApplicationContext的生命周期是伴随整个应用的。
    //而且ApplicationContext每个应用只存在一个，所以就避免了单例模式引用具体的Activity实例
    public static SingleInstance getInstance(Context context) {
        if (instance == null) {
            instance = new SingleInstance(context.getApplicationContext());
        }
        return instance;
    }
```

所以不要把activity、view、context传到周期长的地方去，例如上节课提到的viewModel

最小化变量作用域

尽可能将变量定义到函数类（但是ondraw和getView还是尽量定义在外面吧），因为当一个函数被回收后，里面的变量也会被回收。但是如果变量是定义在类里面的，那么只能等到类被回收了，变量才会被回收。我们也不清楚这期间会发什么（摊手）

总结下性能优化的几种方法

1.避免创建过多的对象（进行大量的异步操作等）

2.不过多使用枚举（内存大）

3.适当使用弱引用和软引用（我们可以打开谢磊学长的课件）

4.采用内存缓存

5.尽量采用静态内部类，避免内部类导致的内存泄漏

6.可以采用SparesArray和Pair等数据结构，性能良好